SAMS Teach Yourself

- 全球销量逾百万册的系列图书
- 连续十余年打造的经典品牌
- 直观、循序渐进的学习教程
- 掌握关键知识的良好起点
- “Read Less, Do More”（精读多练）的教学理念
- 以示例引导读者完成常见的任务

每章内容针对初学者精心设计，1小时轻松阅读学习，
24小时彻底掌握关键知识

每章案例与练习题助你轻松完成常见任务，
通过实践提高应用技能，巩固所学知识

Go语言

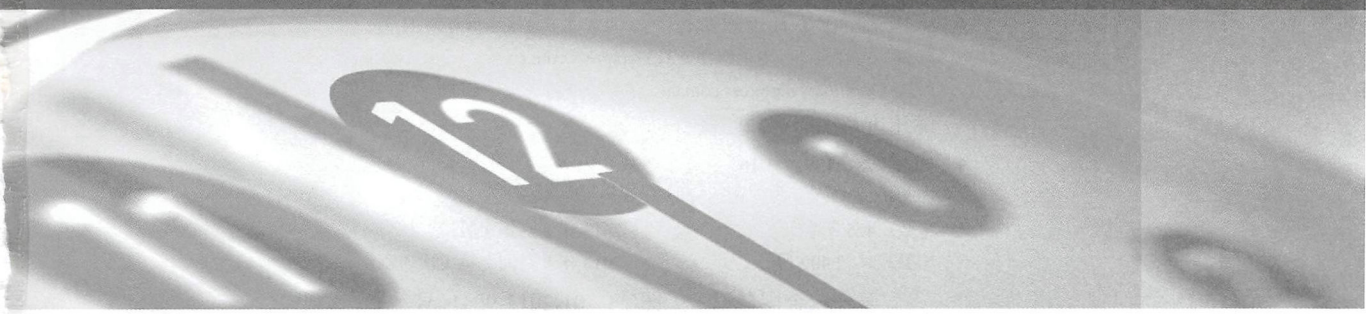
入门经典

[英] 乔治·奥尔波（George Ornbo） 著
张海燕 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

Go语言入门经典

[英] 乔治 · 奥尔波（George Ornbo） 著
张海燕 译



人民邮电出版社
北京

图书在版编目（CIP）数据

Go语言入门经典 /（英）乔治·奥尔波（George Ornbo）著；张海燕译. -- 北京：人民邮电出版社，2018.8
ISBN 978-7-115-48503-8

Ⅰ. ①G… Ⅱ. ①乔… ②张… Ⅲ. ①程序语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2018)第125445号

版权声明

◆ 著　[英] 乔治·奥尔波（George Ornbo）
译　张海燕
责任编辑　陈聪聪
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
涿州市京南印刷厂印刷

◆ 开本：787×1092　1/16
印张：17.5
字数：425 千字　2018 年 8 月第 1 版
印数：1－2 400 册　2018 年 8 月河北第 1 次印刷

著作权合同登记号　图字：01-2017-9041 号

定价：59.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

内容提要

Go 语言是谷歌推出的一种全新的编程语言，旨在不损失应用程序性能的情况下降低代码的复杂性，具有“部署简单、并发性好、语言设计良好、执行性能好”等优势，目前国内诸多 IT 公司均已采用 Go 语言开发项目。

本书分为 24 章，讲解了使用 Go 语言编写高质量程序的方法，其内容涵盖了 Go 语言特性和标准库安装包，Go 与 JavaScript 的对比，Go 命令行工具，Go 中的基本概念（比如类型、变量、函数、控制结构、指针、接口等）、错误处理、Goroutine 和通道、Go 代码测试、使用 Go 编写 HTTP 客户端与服务器、处理 JSON 和文件、部署 Go 代码等。

本书适合想要掌握 Go 语言的零基础读者以及对 Go 语言感兴趣的程序员学习，还可作为高等院校教授 Go 语言课程的教材。

作者简介

George Ornbo 是一位软件工程师、博主和作家，拥有 14 年的软件开发经验，其客户既有初创公司，也有大型企业客户；熟悉众多编程语言、UNIX 和 Web 底层协议。当前供职于伦敦的一家区块链初创公司。

献辞

谨以此书献给 Bea 和 Fin，有你们俩真是我天大的福分！

致谢

感谢 Laura Lewin 和 Pearson 编辑团队给我写作本书的机会。还要感谢 Sheri Replin 所做的出色工作！

感谢 Bala Natarajan 和 Yoshiki Shibata 为本书所做的技术审校工作，感谢他们提出的大量卓越的改进建议。

感谢 Robert Griesemer、Rob Pike 和 Ken Thompson 设计出了 Go 语言。

前　言

Go（Golang）语言是编程语言设计的又一次尝试，是对类 C 语言的重大改进。它让您能够访问底层操作系统，还提供了强大的网络编程和并发编程支持。

Go 语言用途众多，其中包括如下几种。

- 网络编程。
- 系统编程。
- 并发编程。
- 分布式编程。

很多重要的开源项目都是使用 Go 语言开发的，其中包括 Go-Ethereum、Terraform、Kubernetes 和 Docker。Go 语言给开源界带来了重大影响，有望提高开源项目的成功率。

本书读者对象

本书不要求读者具备任何编程或计算机方面的经验，但如果对编程有基本的认识将大有裨益。由于 Go 代码主要是从终端运行的，因此熟悉终端以及如何在其中执行基本命令是十分必要的。最后，鉴于 Go 语言常用于系统编程和网络编程，因此对互联网的工作原理有些了解也将很有帮助，虽然这并非必要的。

为何要学习 Go 语言

如果您要创建系统程序或基于网络的程序，Go 语言是很不错的选择。作为一种相对较新的语言，它是由经验丰富且受人尊敬的计算机科学家设计的，旨在应对创建大型并发网络程序面临的挑战。如果您觉得 Java 或 C 语言的语法导致编程很难，那么 Go 语言将可能提供更佳的体验。对具备诸如 Ruby、Python、JavaScript 等动态语言使用经验的程序员来说，Go 语言提供了类型安全，同时又不像传统语言那么死板。

组织结构

本书首先介绍 Go 语言基础知识，包括搭建 Go 语言编程环境以及运行第一个 Go 程序。接下来介绍一些重要的 Go 语言知识，包括字符串、函数、结构体和方法。您将学会如何使用 Goroutine 和通道，这些是 Go 语言特有的功能，它们避免了并发编程的大部分难题。

然后，您将学习如何调试和测试 Go 语言代码，并学会一些帮助编写独具 Go 语言风格代码的技巧。

接下来，您将学习如何编写基本的命令行程序、HTTP 服务器和 HTTP 客户端，并学习如何处理 JSON 数据和文件。

最后，您将学习正则表达式相关的知识、如何处理时间以及如何将 Go 应用程序部署到生产环境中。

代码示例

本书每章都有多个代码示例，旨在帮助您学习 Go 语言，它们与正文一样重要，强烈建议您在阅读过程中运行这些示例代码。本书的示例代码可在异步社区本书页面中下载。

资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

配套资源

本书提供如下资源：

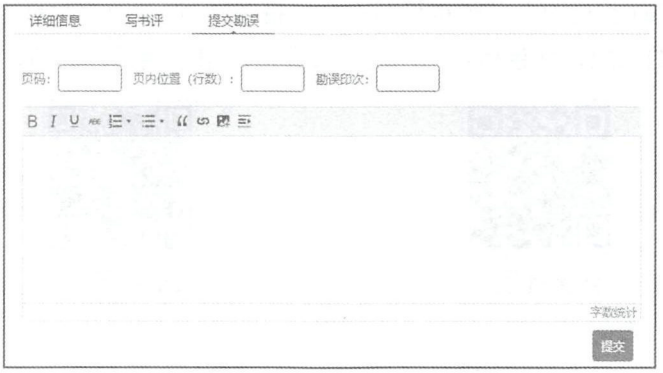
- 本书配套资源请到异步社区的本书购买页面中下载。

要获得以上配套资源，请在异步社区本书页面中点击 配套资源，跳转到下载界面，按提示进行操作即可。注意：为保证购书读者的权益，该操作会给出相关提示，要求输入提取码进行验证。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，点击“提交勘误”，输入勘误信息，单击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的 100 积分。积分可用于在异步社区兑换优惠券、样书或奖品。



扫码关注本书

扫描下方二维码，您将会在异步社区微信服务号中看到本书信息及相关的服务提示。

与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问 www.epubit.com/selfpublish/submission 即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于 2015 年 8 月，提供大量精品 IT 技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网 https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品 IT 专业图书的品牌，依托于人民邮电出版社近 30 年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的 LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

异步社区

微信服务号

目 录

第1章

起步

本章介绍如下内容。

- Go 简介。
- 安装 Go。
- 设置环境。
- 编写您的第一个 Go 程序——Hello World。

通过阅读本章，您将知道 Go 是什么以及创建它的动机是什么。您将学习如何安装 Go 并运行第一个程序。您还将了解 Go Gopher，它会在本书中时不时地蹦出来。

1.1 Go 简介

Go（或 Golang）是 Google 在 2007 年开发的一种开源编程语言，出自 Robert Griesemer、Rob Pike 和 Ken Thompson 之手。2009 年 11 月 10 日，Google Open Source Blog 向全球发布了这款语言；公告指出 Go 的主要目标是“兼具 Python 等动态语句的开发速度和 C 或 C++等编译型语言的性能与安全性”。

1.1.1 Go 语言简史

对语言进行评估时，明白设计者的动机以及语言要解决的问题很重要。Go 语言的设计者都是计算机科学领域的重量级人物。在 20 世纪 70 年代，Ken Thompson 设计并实现了最初的 UNIX 操作系统，仅从这一点说，他对计算机科学的贡献怎么强调都不过分。他还与 Rob Pike 合作设计了 UTF-8 编码方案。除帮助设计 UTF-8 外，Rob Pike 还帮助开发了分布式多用户操作系统 Plan 9，并与人合著了 *The Unix Programming Environment*，对 UNIX 的设计理念做了正统的阐述。Robert Griesemer 就职于 Google，

对语言设计有深入的认识，并负责 Chrome 浏览器和 Node.js 使用的 Google V8 JavaScript 引擎的代码生成部分。

这些计算机科学领域的重量级人物设计 Go 语言的初衷是满足 Google 的需求。设计此语言花费了两年的时间，融入了整个团队多年的经验及对编程语言设计的深入认识。设计团队借鉴了 Pascal、Oberon 和 C 语言的设计智慧，同时让 Go 语言具备动态语言的便利性。因此，Go 语言体现了经验丰富的计算机科学家的语言设计理念，是为全球最大的互联网公司之一设计的。Go 语言的所有设计者都说，设计 Go 语言是因为 C++给他们带来了挫败感。在 Google I/O 2012 的 Go 设计小组见面会上，Rob Pike 是这样说的：

“我们做了大量的 C++开发，厌烦了等待编译完成，尽管这是玩笑，但在很大程度上来说也是事实。”

您无须知道 Go 语言的设计历史就能使用它。您只需知道，Go 语言的设计和实现体现了多位计算机专家多年的经验以及对其他编程语言优缺点的深入认识。因 C++的不良体验而出现的 Go 语言是一门现代编程语言，可用来创建性能卓越的 Web 服务器和系统程序。

1.1.2 Go 是编译型语言

Go 使用编译器来编译代码。编译器将源代码编译成二进制（或字节码）格式；在编译代码时，编译器检查错误、优化性能并输出可在不同平台上运行的二进制文件。要创建并运行 Go 程序，程序员必须执行如下步骤。

1. 使用文本编辑器创建 Go 程序。
2. 保存文件。
3. 编译程序。
4. 运行编译得到的可执行文件。

这不同于 Python、Ruby 和 JavaScript 等语言，它们不包含编译步骤。有关使用编译器的优缺点，将在第 2 章介绍。Go 自带了编译器，因此无须单独安装编译器。

1.2 安装 Go

Go 可用于 FreeBSD、Linux、Windows 和 macOS 等操作系统。如果您使用的是这些操作系统的较新版本，它们很可能支持 Go。有关对这些平台的要求，请参阅 Go 网站列出的系统需求。推荐尽可能使用操作系统自带的包管理器来安装 Go。

开发者也可使用从 Golang 网站下载的文件来安装 Go。如果您是新手，推荐使用安装程序来安装，尽管没有用于 Linux 系统的 Go 安装程序，但通常可使用包管理器来安装；如果您是经验丰富的 Linux 用户，可使用源代码来安装 Go，如果您使用的是 Windows 或 macOS 系统，只需下载相应的文件并双击它，再按说明完成安装过程即可。您将看到一个标准的安装程序窗口，如图 1.1 所示。

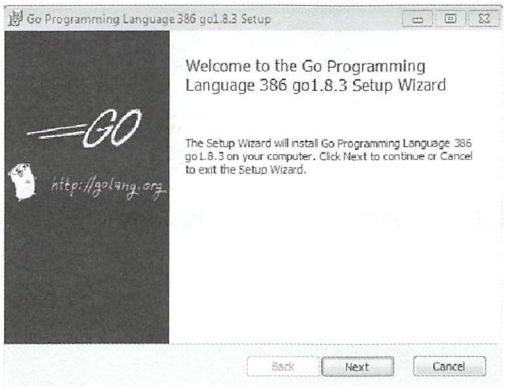


图 1.1

在 Windows 系统中运行 Go 安装程序

1.2.1 在 Windows 系统中安装

安装程序运行完毕后，打开命令提示符并执行命令 go version。如果您看到包含版本号的输出（如图 1.2 所示），就说明正确地安装了 Go。

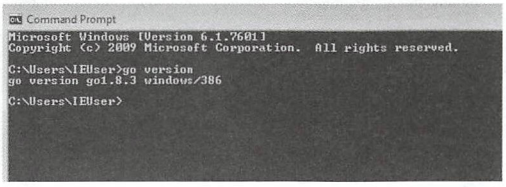


图 1.2

在 Windows 系统中核实正确地安装了 Go

要完成安装，还需为 Go 项目创建目录结构。为此，使用“开始”菜单打开命令提示符，并创建如下目录（如图 1.3 所示）。

```
mkdir %USERPROFILE%\go
mkdir %USERPROFILE%\go\bin
mkdir %USERPROFILE%\go\pkg
mkdir %USERPROFILE%\go\src
```

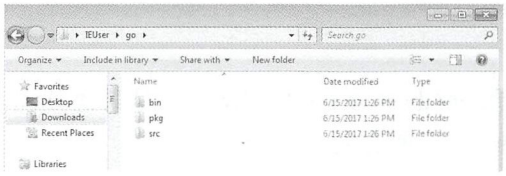


图 1.3

为 Go 项目创建目录结构

创建这些文件夹后，就可添加环境变量 GOPATH 了。在 Windows 系统中添加这个环境变量的步骤如下。

1. 单击“开始”菜单中的“控制面板”。
2. 搜索“环境变量”。
3. 单击“编辑账户的环境变量”，打开如图 1.4 所示的对话框。
4. 单击“New（新建）”按钮。
5. 在 Variable（变量名）文本框中输入 GOPATH。
6. 在 Value（值）文本框中输入%USERPROFILE%\go。
7. 单击“保存”按钮。

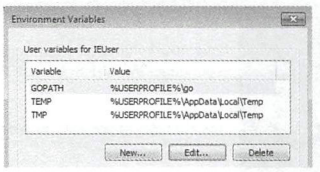


图 1.4

在 Windows 系统中设置环境变量 GOPATH

如果您打开了命令提示符，将其关闭再重新打开。在提示符下执行如下命令，以检查是否设置了环境变量 GOPATH：

```
echo %GOPATH%
C:\Users\george\go
```

1.2.2 在 macOS 或 Linux 系统中安装

安装程序运行完毕后，打开命令提示符并执行命令 go version。如果您看到包含版本号的输出（如图 1.5 所示），就说明正确地安装了 Go。

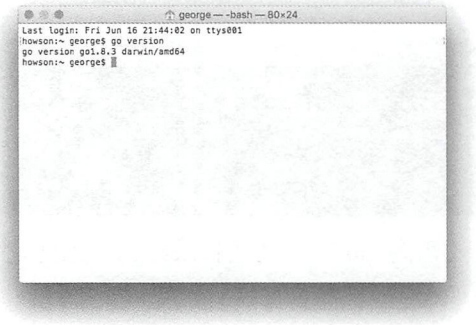


图 1.5

在 macOS 系统中核实是否正确地安装了 Go

要完成安装，还需要为 Go 项目创建目录结构。为此，打开终端并创建如下目录：

```
mkdir $HOME/go
mkdir $HOME/go/bin
mkdir $HOME/go/pkg
mkdir $HOME/go/src
```

为设置环境变量 GOPATH，请编辑.bashrc（如果您运行的是其他 shell，请编辑相应的文件），在其中添加如下内容。.bashrc 位于您的主（home）目录下。

```
export GOPATH=$HOME/go
```

重新加载 shell 或关闭并重新打开终端。在提示符下执行如下命令，以检查是否设置了环境变量 GOPATH：

```
echo $GOPATH
/home/george/go
```

1.3 设置环境

Go 根据您的环境配置行事。作为一个开源项目，代码分享也很重要。为在以后能够分享代码，推荐您创建一个 Github 账户。如果您没有这样的账户，则可前往 Gifhub 网站免费创

建；如果您要使用其他服务，如 Gitlab 或 Bitbucket，则需要在后面的命令中将 github.com 替换为您要使用的服务的 URL。如果您不熟悉 Git 和 GitHub，也不用担心，本书后面会介绍。创建 Github 账户后，像下面这样创建用于存储源代码的文件夹：

```
// Linux / macOS
mkdir -p $GOPATH/src/github.com/[your github username]

// Windows
mkdir %GOPATH%\src\github.com\[your github username]
```

1.4 编写第一个 Go 程序——Hello World

现在该来编写著名的 Hello World 程序了！如果您已安装好 Go，则只要有文本编辑器就可以创建 Go 程序了。如果您还没有喜欢的文本编辑器，可考虑使用 Sublime、Textmate、Notepad++、Atom、Vim、Emacs 等。至于哪款文本编辑器最好，程序员们就此争论得非常激烈，但如果您是新手，选择您用得顺手的编辑器就好。Go 对文本编辑器没有特殊要求，因此如果您只是想尝试一下 Go，使用操作系统提供的默认文本编辑器就可以了。

程序清单 1.1 列出了 Go 程序 Hello World 的代码。这个程序很简单，只是向终端打印一行文本。

程序清单 1.1 Hello World

```
1:  package main
2:
3:  import (
4:      "fmt"
5:  )
6:
7:  func main() {
8:      fmt.Println("Hello World!")
9:  }
```

TRY IT YOURSELF

显示 Hello World!

在这个示例中，您将运行第一个 Go 程序！

1. 创建一个用于存储这个程序的文件夹，再切换到这个文件夹。

```
// Linux / macOS
mkdir -p $GOPATH/src/github.com/[your github username]/hello
cd $GOPATH/src/github.com/[your github username]/hello

// Windows
mkdir %GOPATH%\src\github.com\[your github username]\hello
cd %GOPATH%\src\github.com\[your github username]\hello
```

2. 打开本书代码示例中的文件 hour01/example01。
3. 在刚创建的文件夹 hello 中，创建一个名为 main.go 的文件。

4. 将文件 hour01/example01 的内容复制到 main.go 中。
5. 在文件夹 hello 中，执行命令 go build。
6. 在文件夹中，执行命令 run./main（Linux 或 macOS 系统）或 main.exe（Windows 系统）。
7. 您将在控制台中看到 Hello World!。
8. 祝贺您运行了第一个 Go 程序！

▲

Did you Know?

提示：大多数语言都提供了 Hello World 示例。
Hello World 是一个简单的编程示例，用于展示编程语言的语法。它常用于检查安装是否正确以及简单地介绍语言。它还能够让您对不同的编程语言进行快速比较。

1.4.1 使用 go run 编译并运行程序

编译并运行文件是开发过程中的一个常见步骤，Go 提供了完成这个步骤的快捷途径。使用下面的命令编译并运行程序：

```
go run main.go
```

不同于 go build，go run 不会创建可执行文件。在开发 Go 代码时，go run 提供了一种便利的方式，因为没有必要将编译和执行步骤分开。另外，也不需要运行 go clean 来清除可执行文件。

1.4.2 Go 吉祥物

Go 编程语言有一个吉祥物！在会议、文档页面和博文中，大多会包含如图 1.6 所示的 Go Gopher，这是才华横溢的插画家 Renee French 设计的，她也是 Go 设计者之一 Rob Pike 的妻子。

图 1.6
Go 吉祥物 Gopher（由 Renee French 设计）

1.5 小结

本章介绍了 Go 语言的设计者以及它们设计 Go 语言的动机。您安装了 Go 并运行了第一个程序；还了解了命令 go build 和 go run 的差别以及 Go Gopher。在一章内学习了这么多内容真的很不错！

1.6 问与答

问：既有的语言很多，为何还要设计 Go 语言？

答：设计 Go 语言是因为 Java 和 C++ 等传统语言繁琐、缓慢而难以理解。Go 语言的设计者借鉴了 Python 动态类型语言的优点，旨在打造一款易于使用并可用于开发高流量生产系统的语言。

问：编译器生成的可执行文件很大，但源代码文件很小。为何会这样？

答：在编译器生成的二进制文件中，必须包含执行程序所需的一切。这带来的缺点是二进制文件比源代码文件大，但优点是无须安装依赖就能运行程序。

问：该使用命令 go build 还是 go run？

答：在开发阶段，推荐使用命令 go run；程序开发完毕，可以分享时，建议使用 go build。

1.7 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

1.7.1 小测验

1. 要运行 Go 程序，都需要什么？
2. 您能说出 go build 和 go run 的不同之处吗？
3. Go Gopher 是什么？

1.7.2 答案

1. 除计算机外，您只需终端和文本编辑器。
2. 命令 go build 执行编译，生成一个可执行的二进制文件，这个文件可用来运行程序；命令 go run 编译并运行程序。
3. Go Gopher 是 Go 语言的吉祥物，其身影在整个 Go 生态系统中随处可见。

1.8 练习

1. 修改程序 Hello World，让它显示您想说的其他文本。使用命令 go build 编译这个程序，再运行生成的可执行文件。再次修改要显示的文本，并使用命令 go run 运行这个程序。您现在明白这两个命令的差异了吗？
2. 在网上搜索使用其他各种语言编写的 Hello World 程序。您能发现这些程序与本章的示例程序有什么相同之处吗？
3. 如果有时间，请观看 Rob Pike 在 2009 年介绍 Go 编程语言的视频，其中概述了 Golang 的众多设计目标。

第2章

理解类型

本章介绍如下内容。

- 数据类型是什么？
- 区分静态类型和动态类型。
- 使用布尔类型。
- 理解数值类型。
- 检查变量的类型。
- 类型转换。

Go 是一种静态类型语言，而静态类型是一个必须理解的概念，如果您不是学计算机的，或者使用过 Python、JavaScript 等动态语言，这尤其重要。本章介绍数据类型，并帮助您理解强类型语言和动态类型语言的差别。您将学习 Go 语言类型系统的工作原理及其基本类型。

2.1 数据类型是什么

数据类型让编程语言、编译器、数据库和代码执行环境知道如何操作和处理数据。例如，如果数据类型为数字，通常可对其执行数学运算。编程语言和数据库常常根据数据类型赋予程序员不同的功能和性能。大多数编程语言还提供了用于处理常见数据的标准库，而数据库提供了查询语言，让程序员能够根据底层数据类型来查询数据以及与之交互。无论数据类型是否被显式地声明，它们都是重要的编程和计算结构。

2.2 区分静态类型和动态类型

所谓强类型语言，指的是错误地使用了类型时，编译器将引发错误；所谓动态类型（也叫松散类型或弱类型）语言，指的是为了执行程序，运行时会将一种类型转换为另一种类型，

或者编译器没有实现类型系统。哪种语言更好呢？这存在很大的争议，计算机科学家看重强类型语言的正确性和安全性，而其他人则看重动态语言的简单性和开发速度。

下面是静态类型语言的一些优点。

- 性能高于动态类型语言。
- Bug 通常会被编译器发现。
- 代码编辑器可提供代码补全和其他功能。
- 数据完整性更好。

下面是动态类型语言的一些优点。

- 使用动态类型语言编写软件的速度通常更快。
- 无须为执行代码而等待编译器完成编译。
- 动态类型语言通常不那么死板，因此有些人认为变更代码更容易。
- 有些人认为动态类型语言学习门槛更低。

在 Go 中，程序员可显式地声明类型，也可让编译器推断类型。在本章中，我们将显式地声明类型，以帮助您理解。程序清单 2.1 是一个向终端打印消息的程序。

程序清单 2.1 类型简介

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func sayHello(s string) string {
 8:      return "Hello " + s
 9:  }
10:
11:  func main() {
12:      fmt.Println(sayHello("George"))
13:  }
```

如果您不能完全理解程序清单 2.1，也不用担心。这里的重点是函数 sayHello 的声明。从参数声明可知，这个函数接受一个类型为 string 的参数；这个函数的返回值也是字符串。因此，编译这个程序时，编译器将检查传递给这个函数的参数是否是字符串；如果不是，编译器将引发错误。这正是我们希望的，因为这意味着错误可能根本不会让用户遇到。

为了比较强类型语言和动态类型语言，下面来看一个 JavaScript 示例。JavaScript 是一种使用广泛的动态类型语言，如果您不熟悉，也没关系。这里的重点是看看它在处理类型方面有何不同。

程序清单 2.2 是一个简单的 JavaScript 函数，它接受两个值，将它们相加并返回结果。

程序清单 2.2 简单的 JavaScript 函数

```
1:  var addition = function (a, b) {
2:      return x + y;
3:  };
```

给这个函数提供两个数字时，它能够正确地运行。

```
addition(1,3)
4
```

然而，如果向它传递一个数字和一个字符串，结果将很怪异。

```
addition(1,"three")
1three
```

在这种情况下，这个函数返回一个字符串。怎么会这样呢？虽然 JavaScript 有类型的概念，但其类型使用规则非常宽松。在这个示例中，JavaScript 对数字值执行类型转换，将其转换为字符串，因此返回字符串 1three。JavaScript 提供的这种灵活性虽然很有吸引力，但可能导致微妙乃至灾难性的 Bug。

假设有个程序使用上述 addition 函数来接受输入并将其存储到数据库中。数据库通常有数据类型的概念，很多还有整数的概念。整数是没有小数部分的数字，可正可负。如果数据库将字段定义成了整型，就会要求提供给该字段的值为整数。

前述 JavaScript 函数 addition 可能返回一个字符串，也可能返回一个整数。如果传递给它的值至少有一个字符串，返回的就是字符串。如果这个返回值被插入到需要整数的数据库字段中，将引发错误。更糟糕的是，这种错误发生在运行阶段，这意味着它将影响使用程序的用户。这种错误除非得到妥善处理，否则可能导致程序崩溃。

而在使用 Go 语言编写的函数中，对参数和返回值的类型都做了声明。

程序清单 2.3 是一个使用 Go 语言编写的函数，这个函数指出它接受两个 int 值，并返回一个 int 值。即便不看这个函数的实现，也知道它接受两个整数并返回一个整数。如果程序员错误地将一个字符串传递给这个函数，编译器将捕获这种错误。

程序清单 2.3 Go 语言中的类型

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func addition(x int, y int) int {
 8:      return x + y
 9:  }
10:
11:  func main() {
12:      fmt.Println(2,4)
13:  }
```

在程序清单 2.4 所示的示例中，向函数传递了类型不正确的参数。在这个示例中，将在编译阶段发现这种错误，避免用户遭遇软件崩溃。

程序清单 2.4 传递类型不正确的参数

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func addition(x int, y int) int {
```

```
 8:      return x + y
 9:  }
10:
11:  func main() {
12:      var s string = "three"
13:      fmt.Println(addition(1, s))
14:  }
```

如果您尝试运行这个程序，将出现编译错误。编译错误提供了有用的信息，让您知道出现错误的原因。在这里，原因是在需要 int 的地方使用了字符串。

```
cannot use s (type string) as type int in argument to addition
```

Go 编译器还能捕获其他常见的错误，如传递的参数太多或太少。

TRY IT YOURSELF ▼

传递类型不正确的参数

在这个示例中，您将明白 Go 编译器是如何实现类型系统的。

1. 打开本书代码示例中的 hour02/example04.go。
2. 在终端执行命令 go run example04.go 来运行这个程序。
3. 您将看到如下错误消息。

```
./example04.go:11: cannot use s (type string) as type int in argument to addition
```

▲

2.3 使用布尔类型

对类型有了基本认识后，就可探索 Go 是如何实现一些基本数据类型的了。首先来看布尔类型。布尔值只能为 true 或 false。虽然有些语言允许使用值 1 和 0 来表示 true 和 false，但 Go 语言不允许。可像下面这样声明布尔变量。

```
var b bool
```

如果没有给布尔变量赋值，它将默认为 false。程序清单 2.5 声明了一个布尔变量，再将其打印到终端。

程序清单 2.5 声明布尔变量

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var b bool
 9:      fmt.Println(b)
```

```
10:  }
```

布尔变量可在声明后重新赋值（参见程序清单 2.6），它们是很有用的编程元素。

程序清单 2.6 给布尔变量重新赋值

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var b bool
 9:      fmt.Println(b)
10:      b = true
11:      fmt.Println(b)
12:  }
```

TRY IT YOURSELF

声明布尔变量并给它重新赋值

在这个示例中，您将明白如何初始化布尔变量以及给它重新赋值。

1. 打开本书代码示例中的 hour02/example06.go。
2. 在终端中执行命令 go run example06.go 以运行这个程序。
3. 您将看到给其中的布尔变量重新赋值了。

```
false
true
```

2.4 理解数值类型

对编程来说，数值不可或缺。然而，如果您没有计算机科学或数学方面的知识，可能对有些术语感到迷惑。您可能听说过浮点数、整数、无符号整数、8 位、64 位、bigint、smallint、tinyint，这些都是整型（数值）类型。要明白这些术语的含义，必须知道数字在计算机内部是以二进制位的方式存储的。二进制位就是一系列布尔值，取值要么为 1，要么为 0。1 位可表示 1 或 0，对于 4 位整数，表 2.1 对其二进制表示和十进制表示做了比较。从该表可知，4 位可表示 16 个不同的数字。

表 2.1 4 位的无符号整数

二进制	十进制
0000	0
0001	1
0010	2
0011	3

续表

二进制	十进制
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

2.4.1 带符号整数和无符号整数

对于带符号整数，需要使用一位来表示符号，这通常是符号-。表 2.1 列出了无符号的 4 位整数，其取值范围为 0～15。带符号整数可正可负，因此 4 位带符号整数的取值范围为-8 到 7，如表 2.2 所示。

表 2.2 4 位的带符号整数

二进制	十进制
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	−8
1001	−7
1010	−6
1011	−5
1100	−4
1101	−3
1110	−2
1111	−1

提示：可使用的最大数值取决于计算机处理器。

在计算机中，可使用的最大数值取决于计算机处理器的体系结构。当今的计算机大多是 64 位的，支持的最大无符号整数为 9223372036854775807。第一代微处理器有些只有 4 位，这意味着只能表示-8～7 的带符号整数！

Did you Know?

在 Go 语言中，声明整型变量的方式如下。

```
var i int = 3
```

类型 int 表示带符号整数，因此可正可负。根据计算机的底层体系结构，int 可能是 32 位的带符号整数，也可能是 64 位的带符号整数。除非要处理的数字非常大，或者非常在乎性能，否则只需使用 int，而无须关心编译器是如何做的。

2.4.2 浮点数

浮点数是带小数点的数字，如 11.2、0.1111、43.22。整数不能包含小数部分，因此要处理分数，必须使用浮点数。根据实际数字的大小，Go 语言中的浮点数可以是 32 位的，也可以是 64 位的。在大多数现代计算机中，推荐使用 float64。

```
var f float32 = 0.111
```

2.4.3 字符串

字符串可以是任何字符序列，其中的字符可能是数字、字母和符号。下面是一些简单的字符串。

- cow。
- $^%$。
- a1234。

几乎所有的编程语言都支持字符串，它们通常包含数字、字母或符号。在 Go 语言中，声明并初始化字符串变量很简单。

```
var s string = "foo"
```

字符串变量可以为空，这种变量非常适合用来累积其他变量中的数据以及存储临时数据。

```
var s string = ""
```

创建字符串变量后，可将其与其他数据相加，但不能修改原来的值。下面的代码创建一个空字符串，再将字符串 foo 附加到末尾，这在 Go 语言中是合法的。

```
var s string = ""
s += "foo"
```

不能对字符串执行数学运算，即便它看起来像个数字。要想对看起来像数字的字符串执行数学运算，必须先将其转换为数字类型。

2.4.4 数组

数组是几乎所有编程语言都支持的另一种数据类型，这是一种比较复杂的类型，因为它包含一系列元素。例如，要表示乐队的成员，使用字符串数组是不错的选择。声明数组时，必须指定其长度和类型。

```
var beatles [4]string
```

在这个示例中，方括号内的数字表示数组的长度，而紧跟在方括号后的是数组的类型——

这里为字符串。

```
beatles[0] = "John"
beatles[1] = "Paul"
beatles[2] = "Ringo"
beatles[3] = "George"
```

数组将在第 6 章详细介绍。

注意： 数组索引从 0 开始。
您可能注意到了，在上述变量声明中，指定的数组长度为 4，但访问这个数组的元素时，索引最大为 3。这是因为在所有数组中，索引都从 0 开始。刚接触时，您可能对这一点感到迷惑。

By the Way

2.5 检查变量的类型

有此情况下，需要检查变量的类型，为此可使用标准库中的 reflect 包，它让您能够访问变量的底层类型。在大多数情况下，编译器都能发现类型不正确的情形，但在调试或必须核实底层类型时，reflect 包将很有用，如程序清单 2.7 所示。

程序清单 2.7 使用 reflect 包检查变量类型

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5:     "reflect"
 5: )
 6:
 7: func main() {
 8:     var s string = "string"
 9:     var i int = 10
10:     var f float32 = 1.2
11:
12:     fmt.Println(reflect.TypeOf(s))
13:     fmt.Println(reflect.TypeOf(i))
14:     fmt.Println(reflect.TypeOf(f))
15: }
```

TRY IT YOURSELF ▼

检查类型

1. 打开本书代码示例中的 hour02/example07.go。
2. 在终端中执行命令 go run example07.go 以运行这个程序。
3. 控制台中将显示了变量的类型。

```
string
int
float32
```

▲

2.6 类型转换

将数据从一种类型转换为另一种类型是常见的编程任务，这通常是在从网络或数据库读取数据时进行的。Go 标准库提供了良好的类型转换支持。strconv 包提供了一整套类型转换方法，可用于转换为字符串或将字符串转换为其他类型。

假设有一个字符串变量 s，其值为“true”，要将其用于布尔比较，必须先转换为布尔类型。

```
var s string = "true"
b, err := strconv.ParseBool(s)
```

变量 b 的类型为布尔值。同样，布尔值也可转换为字符串。

```
s := strconv.FormatBool(true)
fmt.Println(s)
```

程序清单 2.8 是一个将布尔值转换为字符串的示例。

程序清单 2.8 类型转换和类型检查

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "strconv"
 6:      "reflect"
 7:  )
 8:
 9:  func main() {
10:      var b bool = true
11:      fmt.Println(reflect.TypeOf(b))
12:      var s string = strconv.FormatBool(true)
13:      fmt.Println(reflect.TypeOf(s))
14:  }
```

▼TRY IT YOURSELF

类型转换

在这个示例中，您将明白如何使用 strconv 包来执行类型转换。

1. 在文本编辑器中打开文件 hour02/example08.go，并尝试理解这个示例是做什么的。
2. 在终端中执行命令 go run example08.go 以运行这个程序。
3. 您将在控制台中看到显示的类型，这表明已经将布尔数据转换成了字符串。

```
bool
string
```

▲

Did you Know?

提示： 理解数据结构。

在编程中，错误地理解数据类型通常会导致 Bug 出现。使用数据源前，请先花点时间搞明白数据类型。如果数据源为数据库，请了解数据库模式（schema）及其使用的数据类型，这将节省大量的调试时间！

2.7 小结

本章简要地介绍了数据类型。您知道了强类型语言和动态类型语言的差别以及它们的优缺点，了解了 Go 语言中一些基本数据类型，学习了如何检查变量的数据类型以及如何进行类型转换。本章介绍的内容很多，而且包含了一些基本的计算知识。

2.8 问与答

问：字符串“1234”为何是字符串，而不是数字？

答：不管字符串看起来有多像数字，它都是字符串。要将其作为数字使用，必须先转换为数字类型。

问：数组能存储不同类型的数据吗？

答：能。有关这样的示例，请参阅第 6 章内容。

问：我熟悉 C 等其他语言。请问在 Go 语言中，我需要负责分配内存吗？

答：Go 负责分配内存和执行垃圾收集。虽然正常的内存使用规则依然适用，但您无须直接管理内存。

2.9 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

2.9.1 小测验

1．强类型语言有何优缺点？

2．带符号整数和无符号整数有何不同？

3．如何检查变量的类型？

2.9.2 答案

1．强类型语言的数据完整性更高，且编译器通常能够在代码执行前发现 Bug。与使用动态类型语言相比，使用强类型语言时，开发速度可能慢一些，条条框框也多一些。鉴于强类型语言过于严格且编译速度缓慢，Go 语言应运而生，旨在兼具动态类型语言的速度与灵活性和静态类型语言的性能与完整性。

2．无符号整数没有占据位的符号。4 位无符号整数的可能取值为 16 个，只能为正；4 位带符号整数的可能取值也是 16 个，但可正可负。

3．要检查变量的类型，可使用 reflect 包。要了解 reflect 包，请参阅程序清单 2.6。

2.10 练习

1．列举本章介绍的数字类型。阐述不同数据类型的差别，如果必要请参阅本章前面的内容。

2．创建一个简短的程序，将字符串变量和 int 变量之间进行转换。

3．如果有时间，请观看 Gary Bernhardt 的讲解视频 WAT。该视频谈论了包括 JavaScript 在内的动态类型语言。切忌不分青红皂白地全盘接受！

第 3 章

理解变量

本章介绍如下内容。

- 变量是什么？
- 快捷变量声明。
- 理解变量和零值。
- 编写简短变量声明。
- 哪种变量声明方式更好？
- 变量作用域。
- 使用指针。
- 声明常量。

变量是计算机程序不可或缺的部分。本章介绍如何在 Go 程序中创建和使用变量，还将介绍 Go 语言中的变量声明方式和变量作用域。

3.1 变量是什么

如果您使用过其他语言进行编程，就不会对变量感到陌生。变量就是值的引用，是实现程序逻辑的基石之一。在 Go 语言中，声明变量的方式有多种。第 2 章介绍过，Go 是一种静态类型语言，因此声明变量时必须显式或隐式地指定其类型。在程序清单 3.1 中，声明了一个名为 s 的变量，其类型为 string。

程序清单 3.1　声明 string 变量

```
1:  package main
2:
3:  import (
4:      "fmt"
5:  )
6:
```

```
 7:  func main() {
 8:      var s string = "Hello World"
 9:      fmt.Println(s)
10:  }
```

对程序清单 3.1 解读如下。

1. 使用关键字 var 声明一个变量。
2. 这个变量名为 s。
3. 这个变量的类型为 string。
4. 赋值运算符=表示将它右边的值赋给变量。
5. 将字符串字面量 Hello World 赋给变量 s。
6. 标准库中的 fmt 包通过变量 s 来引用其值，并将这个值传递给方法 PrintLn。
7. 打印 s 的值。

程序清单 3.1 中，在声明变量的同时给它赋值，但也可在声明变量后再给它赋值，如程序清单 3.2 所示。

程序清单 3.2 声明变量后再给它赋值

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6
 7:  func main() {
 8:      var s string
 9:      s = "Hello World"
10:      fmt.Println(s)
11:  }
```

第 2 章说过，变量的类型很重要，因为这决定了可将什么值赋给变量。例如，对于类型为 string 的变量，不能将整数值赋给它；同理，不能将字符串赋给布尔变量。将类型不正确的值赋给变量时，将导致编译错误。在程序清单 3.3 中，将字符串赋给了类型为 int 的变量。

程序清单 3.3 将类型不正确的值赋给变量

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var i int
 9:      i = "One"
10:      fmt.Println(i)
11:  }
```

运行这个示例将出现编译阶段错误，因为其中的代码试图将字符串赋给整型变量。单词 One 不是整数，不能将其赋给整型变量。

```
go run example03.go
# command-line-arguments
```

```
./example03.go:7:  cannot use "One" (type string) as type int in assignment
```

3.2 快捷变量声明

Go 支持多种快捷变量声明方式。可在一行内声明多个类型相同的变量并给它们赋值，如程序清单 3.4 所示。

程序清单 3.4 快捷变量声明

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var s, t string = "foo", "bar"
 9:      fmt.Println(s)
10:      fmt.Println(t)
11:  }
```

对于不同类型的变量，可使用程序清单 3.5 所示的语法来声明。

程序清单 3.5 以快捷方式声明类型不同的变量

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var (
 9:          s string = "foo"
10:          i int = 4
11:      )
12:      fmt.Println(s)
13:      fmt.Println(i)
14:  }
```

声明变量后，就不能再次声明它。虽然可以给变量重新赋值，但不能重新声明变量，否则将导致编译阶段错误。

```
var s int = 1
fmt.Println(s)
// This is not permitted
var s string = "Hello World"
```

3.3 理解变量和零值

在 Go 语言中，声明变量时如果没有给它指定值，则变量将为默认值，这种默认值被称为零值。这不同于其他语言，因为在这些语言中，未赋值的变量的值为 nil 或 undefined。程序清单 3.6 演示了变量的默认值。变量的默认值取决于其类型。

程序清单 3.6 变量的零值

```
 1:  package main
```

```
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var i int
 9:      var f float
10:      var b bool
11:      var s string
12:      fmt.Printf("%v %v %v %q\n", i, f, b, s)
13:  }
```

▼TRY IT YOURSELF

变量默认被初始化为零值

在这个示例中，您将明白 Go 如何初始化未指定值的变量。

1. 打开本书代码示例中的 hour03/example06.go。
2. 在终端中执行命令 go run example06.go 运行这个程序。
3. 您将在控制台中看到各个变量的零值。

```
go run example06.go
0 0 false ""
```

▲

使用变量时，知道 Go 语言的这种设计决策很重要。不久后，您就可能需要检查变量是否赋值了。注意，在 Go 语言中，为确定变量是否已经赋值，不能检查它是否为 nil，而必须检查它是否为默认值。由于类型 string 的零值为" "，因此对于类型为 string 的变量，要确定是否已经给它赋值，可检查其值是否为零值" "，如程序清单 3.7 所示。

程序清单 3.7　检查变量的值是否为零值

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var s string
 9:      if s == "" {
10:          fmt.Printf("s has not been assigned a value and is zero valued")
11:      }
12:  }
```

Go 禁止将变量初始化为 nil 值，因为这样做将导致编译阶段错误。

3.4　编写简短变量声明

在函数中声明变量时，可使用更简洁的方式——简短变量声明。程序清单 3.8 演示了简

短变量声明。

程序清单 3.8 简短变量声明

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      s := "Hello World"
 9:      fmt.Println(s)
10:  }
```

对程序清单 3.8 解读如下。

1．声明一个名为 s 的变量。请注意，这里没有指定关键字 var 和类型。

2．简短变量赋值语句:=表明使用的是简短变量声明，这意味着不需要使用关键字 var，也不用指定变量的类型。同时这还意味着应将:=右边的值赋给变量。

3．将字符串字面量“Hello World”赋给变量 s。

使用简短变量声明时，编译器会推断变量的类型，因此您无须显式地指定变量的类型。请注意，只能在函数中使用简短变量声明。

3.5 变量声明方式

您可能注意到了，Go 提供了多种变量声明方式。为完整起见，下面列出了所有的变量声明方式。

```
var s string = "Hello World"
var s = "Hello World"
var t string
t = "Hello World"
u := "Hello World"
```

该使用哪种方式呢？Go 对此有一定的限制——不能在函数外面使用简短变量声明。在遵守这条规则的前提下，怎么做都可以。

当然，如何声明变量是风格问题。在同一行内声明变量并给它赋值时，Go 语言设计者在标准库中遵循的约定如下：在函数内使用简短变量声明，在函数外省略类型。程序清单 3.9 演示了这种被普遍接受的约定。如果您查看 Go 源代码，将发现简短变量声明是最常用的变量声明方式。

程序清单 3.9 惯用的变量声明

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  var s = "Hello World"
 6
 7:  func main() {
```

```
 8:       i := 42
 9:       fmt.Println(s)
10:       fmt.Println(i)
11:   }
```

▼ TRY IT YOURSELF

惯用的变量声明

在这个示例中，您将明白 Go 语言惯用的变量声明方式。

1. 打开本书代码示例中的 hour03/example09.go。尝试理解各种不同的变量声明方式。
2. 在终端中使用命令 go run example09.go 运行这个程序。
3. 您将在控制台中看到各个变量的值。

```
Hello World
42
```

▲

3.6 理解变量作用域

术语作用域指的是变量在什么地方可以使用，而不是变量是在什么地方声明的。Go 语言使用基于块的词法作用域。乍一看，术语词法作用域令人望而生畏，但您必须掌握其中的原理。词法是一个形容词，意思与语言的词汇表相关。这意味着 Go 定义了变量在什么地方可以引用，在什么地方无法引用。对编程来说，这很有必要，因为这样可根据引用变量的位置，确定引用的是哪个变量。在 Go 语言中，块是位于一对大括号内的一系列声明和语句，但可以是空的。

可使用通俗的语言前述概念做如下诠释。

- 在 Go 语言中，一对大括号（{}）表示一个块。
- 对于在大括号（{}）内声明的变量，可在相应块的任何地方访问。
- 大括号内的大括号定义了一个新块——内部块。
- 在内部块中，可访问外部块中声明的变量。
- 在外部块中，不能访问在内部块中声明的变量。

简而言之，每个内部块都可访问其外部块，但外部块不能访问内部块。

- 程序清单 3.10 显示了大括号定义的程序结构和变量作用域。每对大括号都表示一个块。
- 代码的缩进程度反映了块作用域的层级。在每个块中，代码都被缩进。
- 在内部块中，可引用外部块中声明的变量。

程序清单 3.10 Go 语言中的词法作用域

```
1:  package main
2:
```

```
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  var s = "Hello world"
 8:
 9:  func main() {
10:      fmt.Printf("Print 's' variable from outer block %v\n", s)
11:      b := true
12:      if b {
13:          fmt.Printf("Printing 'b' variable from outer block %v\n", b)
14:          i := 42
15:          if b != false {
16:              fmt.Printf("Printing 'i' variable from outer block %v\n", i)
17:          }
18:      }
19:  }
```

请注意，变量 s 不是在大括号内声明的，但可在内部块中访问它。这是因为 Go 语言将文件也视为块，所以在第一级大括号外声明的变量可在所有块中访问。

TRY IT YOURSELF ▼

理解变量作用域

在这个示例中，您将明白 Go 语言中变量的作用域。

1. 打开本书代码示例中的 hour03/example10.go，尝试理解其中各个变量的作用域。
2. 在终端中使用命令 go run example10.go 运行这个程序。
3. 您将在终端看到作用域各不相同的变量。

```
Printing 's' variable from outer block Hello world
Printing 'b' variable from outer block true
Printing 'i' variable from outer block 42
```

▲

3.7 使用指针

指针是另一个与变量相关且必须掌握的要素。在 Go 语言中声明变量时，将在计算机内存中给它分配一个位置，以便能够存储、修改和获取变量的值。要获取变量在计算机内存中的地址，可在变量名前加上&字符。程序清单 3.11 将一个变量的内存地址打印到终端。

程序清单 3.11 打印变量在内存中的地址

```
1:  package main
2:
3:  import (
4:      "fmt"
5:  )
6:
7:  func main() {
8:      s := "Hello World"
9:      fmt.Println(&s)
```

```
10:  }
```

如果您运行这些代码，将打印一个由字母和数字组成的序列，它表示变量在内存中的地址，而不是变量的值。

TRY IT YOURSELF

使用指针

在这个示例中，您将明白如何使用指针。

1. 打开本书代码示例中的 hour03/example11.go。
2. 在终端中使用命令 go run example11.go 运行这个程序。
3. 您将在控制台中看到一个内存地址。

```
0xc42000e230
```

这是内存地址的十六进制表示。如果您不知道十六进制为何物，也不用担心，您只需知道每个变量的值都存储在不同的内存地址中就够了。在程序清单 3.12 中，将一个整型变量传递给了一个函数，并打印了两个变量的内存地址。您认为它们的内存地址相同吗？

程序清单 3.12 将变量作为值传递

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func showMemoryAddress(x *int) {
 8:      fmt.Println(x)
 9:      return
10:  }
11:
12:  func main() {
13:      i := 1
14:      fmt.Println(&i)
14:      showMemoryAddress(&i)
15:  }
```

这个程序运行时显示两个内存地址。

```
go run example13.go
0xc42000a2e8
0xc42000a2e8
```

将变量传递给函数时，会分配新内存并将变量的值复制到其中。这样将有两个变量实例，它们位于不同的内存单元中。一般而言，这不可取，因为这将占用更多的内存，同时由于存在变量的多个副本，很容易引入 Bug。考虑到这一点，Go 提供了指针。

指针是 Go 语言中的一种类型，指向变量所在的内存单元。要声明指针，可在变量名前

加上星号字符。可将前面的示例修改成如程序清单 3.13 所示的代码，以使用指针。

程序清单 3.13 将变量作为指针传递

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func showMemoryAddress(x *int) {
 8:      fmt.Println(x)
 9:      return
10:  }
11:
12:  func main() {
13:      i := 1
14:      fmt.Println(&i)
14:      showMemoryAddress(&i)
15:  }
```

这个程序运行时显示两个内存地址。

```
go run example13.go
0xc42000a2e8
0xc42000a2e8
```

对代码的修改情况说明如下。

- 将传递给 showMemoryAddress 的值从 i 改成了&i。i 前面的和号&意味着引用的是变量 i 的值所在的内存地址。
- 将函数 showMemoryAddress 的第一个参数的类型从 int 改成了*int。加上星号意味着参数的类型为指向整数的指针，而不是整数。
- 在函数 showMemoryAddress 中打印变量时，不需要使用和号，因为它本来就是指针。

如果要使用指针指向的变量的值，而不是其内存地址，该怎么办呢？可在指针变量前加上星号。

```
func showMemoryAddress(x *int) {
    fmt.Println(*x)
    return
}
```

使用星号后，打印的是值而不是内存地址。

3.8 声明常量

常量指的是在整个程序生命周期内都不变的值。常量初始化后，可以引用它，但不能修改它。程序清单 3.14 声明并初始化一个常量，再将其打印到控制台。

程序清单 3.14 声明常量

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
```

```
)

const greeting string = "Hello, world"

func main() {
    fmt.Println(greeting)
}
```

在程序中试图修改常量将导致错误。程序清单 3.15 就是一个这样的示例，它试图修改常量的值。

程序清单 3.15 试图修改常量

```
package main

import (
    "fmt"
)

const greeting string = "Hello, world"

func main() {
    greeting = "Goodbye, cruel world"
    fmt.Println(greeting)
}
```

运行程序清单 3.15 将导致错误。

```
go run example15.go
# command-line-arguments
./example15.go:10:  cannot assign to greeting
```

3.9 小结

本章介绍的内容很多。您学习了很多重要的 Go 语言构件，如包括快捷语法在内的变量声明方式。您学习了 Go 语言中的零值以及 Go 语言如何给变量指定默认值。您学习了简短变量声明以及首选的变量声明方式。您学习了如何声明值不变的常量。您学习了作用域这个重要概念，它指的是可在什么地方引用变量。最后，您学习了指针以及变量是如何在内存中存储的。

3.10 问与答

问：在 Go 语言中，有很多声明变量的方式，我怎么知道该使用哪种方式呢？

答：要理解 Go 语言约定，一种不错的方式是查看 Go 本身的源代码。选择您熟悉的包名，并查看其代码中的变量声明。

问：不声明变量的类型是不是很危险？编译器在定义变量的类型时会不会出现错误呢？

答：对于未显式声明类型的变量，Go 编译器很善于动态地推断其类型。如果编译器推断不出来（这种可能性很小），它会告诉您的。

问：我不太熟悉内存，请问有什么资料可让我更深入地了解内存？

答：斯坦福大学教授 Mehran Sahami 的讲座探索了内存工作的原理。虽然该讲座针对的是 Java，但大部分内容也适用于 Go 语言。

3.11 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

3.11.1 小测验

1. 在什么情况下不能使用简短变量声明（:=）？
2. 在 Go 语言中，块由什么表示？
3. 常量有何作用？

3.11.2 答案

1. 在函数外面不能使用简短变量声明。
2. 在 Go 语言中，块由一对大括号表示。另外，还有文件级块、包级块和全局块。
3. 常量是整个程序运行期间都不变的值，如纳税代码和佣金率，在没有重新编译程序的情况下，不应修改这些值。

3.12 练习

1. 编写一个小程序，在其中创建一个字符串变量并打印它；然后修改这个变量的值并再次打印它；最后，将这个变量的值改为整数，如 42。结果将如何呢？为何会这样？
2. 为加深对作用域的理解，请修改示例 10。尝试在外部块中使用内部块中声明的变量，结果如何呢？为何会这样？在内部块中引用一些在外部块中声明的变量，并打印它们。您觉得自己理解了变量作用域了吗？
3. 凭记忆阐述将变量传递给函数时发生的情况。就内存方面而言，这意味着什么？为何应使用指针？

第4章

使用函数

本章介绍如下内容。

- 函数是什么？
- 定义不定参数函数。
- 使用具名返回值。
- 使用递归函数。
- 将函数作为值传递。

函数是另一个核心要素，不仅仅是 Go 语言，在日常编程中亦如此。本章将从返回单个结果的简单函数着手，介绍如何在 Go 语言中使用函数。您将看到，Go 函数可返回多个结果，还可接受数量不定的参数。Go 语言还提供了一些让它看起来像函数式编程语言的功能。

4.1 函数是什么

简单地说，函数接受输入并返回输出。数据流经函数时，将被变换。一个典型的示例是将两个数相加的简单函数，它接受两个数字，将它们相加并返回结果。在这个简单示例中，接受的两个数字被称为输入，相加得到的结果被称为输出。

```
func addUp(x int, y int) int {
    return x + y
}
```

4.1.1 函数的结构

在 Go 语言中，函数向编译器和程序员提供了有关的信息，这些信息指出了函数将接受什么样的输入并提供什么样的输出。这种信息是在函数的第一行中提供的，而这一行被称为函数签名。我们再以函数 addUp 为例，阐述 Go 语言中函数的结构是什么样的。

```
func addUp(x int, y int) int {
    return x + y
}
```

关键字 func 指出这是一个函数的开头位置。接下来是函数名，这是可选的，但能够让您在其他地方调用（或使用）这个函数。接下来是一对括号，指出了函数接受什么样的值，在这里，是两个类型为 int 的值（带符号整数，长度至少为 32 位）。在右括号后面是返回值，这里也是一个类型为 int 的值。左大括号表示接下来为函数体，函数体以右大括号结束。如果函数签名声明了返回值，则函数体必须以终止语句结束。通常有一个返回值，但并非总是如此。

4.1.2 返回单个值

别忘了，在最简单的情况下，函数接受一个输入并返回一个输出，比如判断一个数字是奇数还是偶数的函数。实现函数前，务必花点时间想一想它要做什么、输入值是什么、返回值是什么。这不仅有助于设计函数，还能让您知道如何测试它。

编程语言通常提供了多种解决问题的方式，因此清楚函数的输入和输出后，从设计角度看函数的实现将不那么重要。

对于判断一个数字是奇数还是偶数的函数，可对其做如下假设。

- 这个函数接受一个整型参数。
- 这个函数返回一个布尔值：如果传入的整数为偶数，就返回 true；否则返回 false。

本章前面说过，函数的输入和输出类型是在签名中声明的。只要稍微想一想，就可编写出这个函数的声明。

```
func isEven(i int) bool {
}
```

这个函数名为 isEven，它将一个整数作为输入并返回一个布尔值。确定函数签名后，就可编写实现了。这个函数必须判断一个整数值是否为偶数，并返回一个布尔值。为此，一种办法是使用求模运算符%，这个运算符将一个整数作为左操作数，将其与右操作数相除，并返回余数。现在可以编写这个函数的实现了。

```
Func isEven(i int) bool {
    return i%2 == 0
}
```

在设计函数方面，程序员仅受制于语言和想像力。编写复杂的软件或团队写作开发时，必须对一些重要的考虑因素做到心中有数。

- 让每个函数只做一件事情并把这件事情做好。软件不可避免地要修改，通过结合使用大量简短的函数，可让软件更容易修改。这还有助于测试各个函数以及整个软件。
- 维护。在团队合作开发中，您编写的函数易于阅读和理解吗？如果不是这样的，就说明它过于复杂或必须添加注释。别忘了，您可能在一年后的午夜时分回过头来阅读这个函数！

- 性能。在有些情况下，函数的性能至关重要。定义明确的函数能够让程序员修改其实现，并测试其性能是否达到了目标基准。就函数 isEven 而言，调用者并不关心其实现——只要它实现了签名指定的功能。这让您能够轻松地修改其实现。

程序清单 4.1 声明并调用了一个函数。

程序清单 4.1 调用函数

```
 1:  package main
 2:
 3:  import "fmt"
 4:
 5:  func isEven(i int) bool {
 6:      return i%2 == 0
 7:  }
 8:
 9:  func main() {
10:      fmt.Printf("%v\n", isEven(1))
11:      fmt.Printf("%v\n", isEven(2))
12:  }
```

要调用函数，可通过名称来对其进行引用，并提供所需的参数。在程序清单 4.1 中，调用了函数 isEven 两次，并将结果打印到终端。对函数调用次数没有任何限制。

TRY IT YOURSELF

创建并调用函数

在这个示例中，您将明白如何创建并调用函数。

1. 打开本书代码示例中的 hour04/example01.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example01.go 运行这个程序。
4. 您将在终端中看到如下文本，这表明函数被调用了两次。

```
False
True
```

4.1.3 返回多个值

在 Go 语言中，可在函数签名中声明多个返回值，让函数返回多个结果。在这种情况下，终止语句可返回多个值。在下面的示例中，函数签名指出函数不接受任何参数，并返回一个整数和一个字符串。在函数体中，return 语句返回了多个用逗号分隔的值。

```
func getPrize() (int, string) {
    i := 2
    s := "goldfish"
    return i, s
}
```

调用这个函数时，可直接将返回值赋给变量并使用它们。

```
func main() {
    quantity, prize := getPrize()
    fmt.Printf("You won %v %v\n", quantity, prize)
}
```

程序清单 4.2 演示了一个返回多个结果的函数。

程序清单 4.2 从函数返回多个值

```
 1:  package main
 2:
 3:  import "fmt"
 4:
 5:  func getPrize() (int, string) {
 6:      i := 2
 7:      s := "goldfish"
 8:      return i, s
 9:  }
10:  func main() {
11:      quantity, prize := getPrize()
12:      fmt.Printf("You won %v %v\n", quantity, prize)
13:  }
```

4.2 定义不定参数函数

不定参数函数是参数数量不确定的函数。通俗地说，这意味着它们接受可变数量的参数。在 Go 语言中，能够传递可变数量的参数，但它们的类型必须与函数签名指定的类型相同。要指定不定参数，可使用 3 个点（...）。在下面的示例中，函数签名指定函数可接受任意数量的 int 参数。

```
func sumNumbers(numbers. . .int) int {
```

这个函数可接受一个或多个整数，您可使用它来计算任意多个整数的和并返回单个整数。在这个函数中，变量 numbers 是一个包含所有参数的切片。如果您不知道切片为何物，也不用担心，第 6 章将介绍它。将任意数量的整数相加的函数实现类似于下面的代码。

```
func sumNumbers(numbers. . .int) int {
    total := 0
    for _, number := range numbers {
        total += number
    }
    return total
}
```

您可使用这个函数来计算一系列整数的和。

```
func main() {
    result := sumNumbers(1, 2, 3, 4)
    fmt.Printf("The result is %v\n", result)
}
```

程序清单 4.3 使用了一个不定参数函数来将多个整数相加。

程序清单 4.3 使用不定参数函数

```
 1:  package main
 2:
 3:  import "fmt"
 4:
 5:  func sumNumbers(numbers. . .int) int {
 6:      total := 0
 7:      for _, number := range numbers {
 8:        total += number
 9:      }
10:      return total
11:  }
12:
13:  func main() {
14:      result := sumNumbers(1, 2, 3, 4)
15:      fmt.Printf("The result is %v\n", result)
16:  }
```

▼ TRY IT YOURSELF

理解不定参数函数

在这个示例中，您将明白如何使用不定参数函数。

1. 打开本书代码示例中的 hour04/example03.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example03.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
The result is 10
```

5. 请修改传递给函数的参数数量，并再次运行这个程序。

▲

4.3 使用具名返回值

具名返回值让函数能够在返回前将值赋给具名变量，这有助于提升函数的可读性，使其功能更加明确。要使用具名返回值，可在函数签名的返回值部分指定变量名。

```
func sayHi() (x, y string) {
```

这个签名指定将返回两个值，它们的类型都为 string；它还指定了变量名（x 和 y），您可在函数体中给它们赋值。

```
func sayHi() (x, y string) {
    x = "hello"
    y = "world"
    return
}
```

这个函数体中，在终止语句 return 前给具名变量进行了赋值。使用具名返回值时，无须显式地返回相应的变量。这被称为裸（naked）return 语句。

```
func main() {
    fmt.Println(sayHi())
}
```

调用这个函数时，将按声明顺序返回具名变量。

```
hello world
```

程序清单 4.4 是一个具名返回值使用示例。

程序清单 4.4　具名返回值

```
 1:  package main
 2:
 3:  import "fmt"
 4:
 5:  func sayHi() (x, y string) {
 6:      x = "hello"
 7:      y = "world"
 8:      return
 9:  }
10:
11:  func main() {
12:      fmt.Println(sayHi())
13:  }
```

TRY IT YOURSELF ▼

理解具名返回值

在这个示例中，您将明白如何使用具名返回值。

1. 打开本书代码示例中的 hour04/example04.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example04.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
hello world
```

▲

4.4　使用递归函数

递归函数虽然是一个简单的概念，却也是一个功能强大的编程元素。递归函数是不断调用自己直到满足特定条件的函数。要在函数中实现递归，可将调用自己的代码作为终止语句中的返回值。

```
func feedMe(portion int, eaten int) int {
    eaten = portion + eaten
    if eaten >= 5 {
        fmt.Printf("I'm full! I've eaten %d\n", eaten)
        return eaten
```

```
    }
    fmt.Printf("I'm still hungry! I've eaten %d\n", eaten)
    return feedMe(portion, eaten)
}
```

在这个示例中，最重要的是函数体的最后一行，它没有返回值，而是调用自己，这样将再次执行这个函数。通常，递归函数不断调用自己，直到满足特定条件。在这个示例中，如果变量 eaten 不小于 5，函数将不再调用自己并返回。

程序清单 4.5 演示了一个不断调用自己直到满足特定条件的函数。

程序清单 4.5 递归函数

```
 1:  package main
 2:
 3:  import "fmt"
 4:
 5:  func feedMe(portion int, eaten int) int {
 6:      eaten = portion + eaten
 7:      if eaten >= 5 {
 8:          fmt.Printf("I'm full! I've eaten %d\n", eaten)
 9:          return eaten
10:      }
11:      fmt.Printf("I'm still hungry! I've eaten %d\n", eaten)
12:      return feedMe(portion, eaten)
13:  }
14:
15:  func main() {
16:      fmt.Println(feedMe(1, 0))
17:  }
```

▼ TRY IT YOURSELF

理解递归函数

在这个示例中，您将明白如何使用递归函数。

1. 打开本书代码示例中的 hour04/example05.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example05.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
I'm still hungry! I've eaten 1
I'm still hungry! I've eaten 2
I'm still hungry! I've eaten 3
I'm still hungry! I've eaten 4
I'm full! I've eaten 5
```

▲

4.5 将函数作为值传递

Go 语言提供了一些函数式编程功能，如能够将一个函数作为参数传递给其他函数。这看起

来像《盗梦空间》中的情形，但提供了强大的功能。从本质上说，Go 将函数视为一种类型，因此可将函数赋给变量，以后再通过变量来调用它们。在下面的示例中，将一个函数赋给了变量 fn。

```
func main() {
    fn := func() {
        fmt.Println("function called")
    }
    fn()
}
```

- 使用第 3 章介绍的简短变量赋值运算符将一个函数赋给了变量 fn。
- 声明这个函数并将其定义为打印一行文本，让您知道它被调用了。
- 在变量名 fn 后使用( )调用这个函数。

别忘了，在 Go 语言中，函数是一种类型，因此可将其传递给其他函数。我们可对前一个示例进行扩展，将变量 fn 传递给一个函数，并在这个函数中调用它，如程序清单 4.6 所示。

程序清单 4.6 将函数作为参数传递

```
 1:  package main
 2:
 3:  import "fmt"
 4:
 5:  func anotherFunction(f func() string) string {
 6:      return f()
 7:  }
 8:
 9:  func main() {
10:      fn := func() string {
11:          return "function called"
12:      }
13:      fmt.Println(anotherFunction(fn))
14:  }
```

请注意，函数 anotherFunction 的签名中包含一个子函数签名，这表明这个参数是一个返回字符串的函数。接受函数依然需要声明其返回类型，它可以是任何类型，但这里也是字符串。

TRY IT YOURSELF ▼

理解将函数作为值

在这个示例中，您将明白如何将函数作为值使用。

1. 打开本书代码示例中的 hour04/example07.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example07.go 运行这个程序。
4. 您将在终端中看到如下文本，这表明函数被作为参数被传递给了另一个函数，并在这个函数中被调用。

```
function called
```

▲

4.6 小结

本章介绍了如何在 Go 语言中使用函数。您学习了函数签名和一些基本函数；知道了如何返回单个结果以及多个结果；还学习了如何使用不定参数函数和具名返回值；最后，您学习了如何在函数中调用其本身以及 Go 语言支持将函数作为参数传递给其他函数。

4.7 问与答

问：声明函数后，可修改其参数的数量和类型吗？

答：不能。函数声明后，编译器就记录了其签名，并根据签名检查调用该函数时指定的参数数量和类型是否正确。

问：该使用具名返回值吗？

答：在简短的函数中，可使用具名返回值，但这样可能导致代码阅读起来更困难；在有些情况下，还更容易引入 Bug。编写更啰唆的代码没什么不好，但仅当有助于改善可维护性和可读性时才如此。

问：为何应让函数较短？

答：简短的函数有多个优点。它们更易于理解和测试，还让函数能够专注于做一件事情并把它做好。使用单个庞大的函数可能导致代码成为联系紧密的整体，进而难以修改。

4.8 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

4.8.1 小测验

1. 在 Go 语言中，函数可返回多少个值？
2. 调用自己的函数被称为什么？
3. 在 Go 语言中，可将函数作为参数传递给其他函数吗？

4.8.2 答案

1. 在 Go 语言中，函数可返回一个或多个值。另外，多个返回值的类型可以不同。
2. 调用自己的函数被称为递归函数。
3. 可以。函数被视为一种类型，因此可以传递。在 Go 语言中，函数是一等公民，因为可将其作为参数传递给其他函数。

4.9 练习

1．设计一个将华氏温度转换为摄氏温度的函数（不用编写代码）。这个函数的输入和输出分别是什么？

2．编写一个函数，它调用自己 10 次再退出。

3．编写一个函数，它接受 2 个参数并返回 3 个值。

第5章

控制流程

本章介绍如下内容。

- 使用 **if**、**else** 和 **else if** 语句。
- 使用比较运算符。
- 使用算术运算符。
- 使用逻辑运算符。
- 使用 **switch** 语句。
- 使用 **for** 语句执行循环。
- 使用 **defer** 语句。

本章介绍控制流程以及代码执行流程是如何确定的，让您能够创建以不同的方式响应数据的程序。您将学习 if 语句以及如何使用比较运算符、算术运算符和逻辑运算符来实现逻辑；您将明白如何使用循环来反复多次执行代码块；最后，您将学习 defer 语句，它让您能够在函数结束后执行另一个函数。

5.1 使用 if 语句

if 语句是计算机程序的重要组成部分，几乎所有编程语言都支持。简单地说，if 语句检查指定的条件，并在条件满足时执行指定的操作。“如果……就”范式让程序能够以不同的方式响应数据，还让程序员能够实现在不同条件下采取不同措施的逻辑。无论是在命令行程序还是现代计算机游戏中，都有默默无闻的 if 语句的身影，它支撑着或复杂或简单的逻辑树。您可能想到了，在 Go 语言中，if 语句编写起来很简单，如程序清单 5.1 所示。

程序清单 5.1　if 语句

```
1:  package main
```

```
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      b := true
 9:      if b {
10:          fmt.Println("b is true!")
11:      }
12:  }
```

对程序清单 5.1 解读如下。

- 声明变量 b 并将其初始化为 true。这个变量被推断为布尔类型。
- 使用一条 if 语句判断 b 是否为 true。
- 由于这条 if 语句的结果为 true，因此执行大括号内的代码。
- 在终端中打印一行文本，指出 b 为 true。

TRY IT YOURSELF ▼

使用简单的 if 语句

在这个示例中，您将明白如何使用 if 语句。

1. 复制程序清单 5.1 所示的代码，这些代码也可在本书示例代码中的文件 example01.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example01.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
b is true!
```

▲

如果将 b 的值改为 false，则这条 if 语句的结果将为 false，因此不会执行大括号内的代码，如程序清单 5.2 所示。

程序清单 5.2　结果为 false 的 if 语句

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      b := false
 9:      if b {
10:          fmt.Println("b is true!")
11:      }
12:  }
```

可依次运行多条 if 语句，运行顺序与这些语句在源代码中出现的顺序相同。

if 语句总是计算一个布尔表达式，在它为 true 时执行大括号内的代码，在它为 false 时不执行。简单 if 语句是一种功能强大的范式，仅使用它们就能创建出复杂的程序。

▼TRY IT YOURSELF

理解结果为 false 的 if 语句

在这个示例中，您将明白 if 语句的结果为 false 时将发生的情况。

1. 复制程序清单 5.2 所示的代码，这些代码也可在文件 example02.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example02.go 运行这个程序。
4. 由于 if 语句中的布尔表达式为 false，因此您在终端中什么都看不到。

▲

5.2 使用 else 语句

else 语句指定了到达该分支时将执行的代码。它不做任何判断，只要到达它所在的分支就执行。只要前面有语句的结果为 true，eles 语句就不会执行。在 Go 语言中，else 语句紧跟在其他语句的右大括号后面，通常是当前块中的最后一条语句。大致而言，else 相当于说：如果其他条件都不为 true，就执行这条语句。

else 语句可用来处理前面的 if 语句的结果不为 true 的情形。程序清单 5.3 是一个 else 语句使用示例。

程序清单 5.3 else 语句

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5: )
 6:
 7: func main() {
 8:     b := false
 9:     if b {
10:         fmt.Println("b is true!")
11:     } else {
12:         fmt.Println("b is false!")
13:     }
14: }
```

如果 if 语句的结果为 true，就不会执行 eles 语句。else 语句指定了前面所有语句的结果都为 false 时要执行的代码，只要前面有一条语句的结果为 true，它就将被忽略。

TRY IT YOURSELF ▼

使用 else 语句

在这个示例中，您将明白如何使用 else 语句。

1. 复制程序清单 5.3 所示的代码，这些代码也可在本书示例代码中的文件 example03.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example03.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
b is false!
```

5. 将 b 的值改为 true 再运行这个程序，注意到 else 语句被忽略。

▲

5.3 使用 else if 语句

在很多情况下，都需要依次判断多个布尔表达式，此时可使用 else if 语句。else if 语句能够让您在前面的布尔表达式为 false 时接着判断后面的布尔表达式，这种逻辑的意思是，如果前面的 if 或 else if 语句为 false，就试试这条 else if 语句。else if 语句紧跟在前面的 if 或 else if 语句的右大括号后面，且包含另一个布尔表达式。可依次使用多条 else if 语句。程序清单 5.4 是一个 else if 语句使用示例。

程序清单 5.4 else if 语句

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5: )
 6:
 7: func main() {
 8:     i := 3
 9:     if i == 3 {
10:         fmt.Println("i is 3")
11:     } else if i == 2 {
12:         fmt.Println("i is 2")
13:     }
14: }
```

对程序清单 5.4 解读如下。

- 声明变量 i 并将其初始化为 2。
- 第一条 if 语句判断 i 是否为 3。如果不是，就忽略下一行代码。
- else if 语句使用另一个布尔表达式判断 i 是否为 2。

- 由于 i 为 2，因此执行下一行代码，向终端打印一行文本。

else 和 else if 语句的根本不同在于，else if 语句让您能够判断布尔条件，而 else 语句在到达其所在分支时就会执行。

▼ TRY IT YOURSELF

使用 else if 语句

在这个示例中，您将明白如何使用 else if 语句。

1. 复制程序清单 5.4 所示的代码，这些代码也可在本书代码示例中的文件 example04.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example04.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
i is 2
```

5. 将 i 的值改为 3 并再次运行这个程序，注意到 else if 语句被忽略，因为第一条 if 语句的结果为 true。

▲

5.4 使用比较运算符

布尔表达式只是返回 true 或 false，但使用比较运算符可执行更复杂的评估。比较运算符可用于对任何两项类型相同的数据进行比较，如果它们满足比较运算符指定的条件，结果将为 true；否则为 false。在编程中，常做的一些比较如下。

- 两个字符串是否相同？
- 两个数字是否相同？
- 一个数字是否比另一个数字大？
- 一个数字是否小于等于另一个数字？

表 5.1 列出了 Go 语言支持的所有比较运算符。

表 5.1 比较运算符

字符	运算符
==	等于
!=	不等
<	小于
<=	小于等于
>	大于
>=	大于等于

关于 Go 语言中的比较运算符，一个要点是两个操作数的类型必须相同。例如，无法对字符串和整数进行比较。

5.5 使用算术运算符

在任何对布尔表达式进行评估的语句中，都可使用基本算术运算来改进逻辑和评估。常结合使用算术运算符和比较运算符来创建管理控制流程的布尔表达式。

在布尔表达式中，一些常用的算术和比较运算如下。

- 两个数的和是否等于特定的数字？
- 两个数的差是否大于特定的数字？
- 两个数的商是否等于特定的数字？

表 5.2 列出了 Go 语言中常用的算术运算符。

表 5.2 算术运算符

字符	运算符
+	和（也叫加）
-	差（也叫减）
*	积（也叫乘）
/	商（也叫除）
%	余（也叫模）

与比较运算符一样，算术运算符也只能用于类型相同的操作数。

5.6 使用逻辑运算符

除比较运算符和算术运算符外，逻辑运算符也可用于支持控制流程。逻辑运算符支持 3 种比较。

在布尔表达式中，一些常用的逻辑比较如下。

- 两个变量是否都为 true？
- 在两个变量中，是否至少有一个为 true？
- 两个变量是否都为 true 或都为 false？

表 5.3 列出了 Go 语言中常用的逻辑运算符。

表 5.3 逻辑运算符

字符	运算符
&&	与：两个条件是否都为 true
||	或：两个条件是否至少有一个为 true
!	非：条件是否为 false

5.7 使用 switch 语句

switch 语句可用来替代冗长的 if else 布尔比较，如程序清单 5.5 所示。出于对代码可读性的考虑，很多程序员都喜欢使用 switch 语句（而不使用 if else 语句），从编译层面上说，这样的代码效率也可能更高。除少量的 if else 比较外，其他的都可替换为 switch 语句，这样不仅可提高代码的可读性，还可提高其性能。

程序清单 5.5 switch 语句

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      i := 2
 9:
10:      switch i {
11:      case 2:
12:          fmt.Println("Two")
13:      case 3:
14:          fmt.Println("Three")
15:      case 4:
16:          fmt.Println("Four")
17:      }
18:  }
```

对程序清单 5.5 解读如下。

- switch 语句指定了一个要评估的变量——i。
- 在一系列 case 语句中指定了要与变量 i 比较的表达式。
- 如果这个表达式的结果与 i 相等，就执行相应的语句。
- 如果这个表达式的结果与 i 不等，就跳到下一条 case 语句。
- 在这里，由于 i 为 2，因此向终端打印单词 Two。

TRY IT YOURSELF

使用 switch 语句

在这个示例中，您将明白如何使用 switch 语句。

1. 复制程序清单 5.5 所示的代码，这些代码也可在本书代码示例中的文件 example05.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example05.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
Two
```

5. 将 i 的值改为 3 并再次运行这个程序。您明白 switch 语句的工作原理了吗？

相比于 else if 条件，switch 语句更简洁，它还支持在其他 case 条件都不满足时将执行的 default case。在 switch 语句中，可使用关键字 default 来指定其他 case 条件都不满足时要执行的代码。default case 通常放在 switch 语句末尾，但也可将其放在任何其他地方。程序清单 5.6 演示了如何在 switch 语句中使用 default case。

程序清单 5.6　在 switch 语句中添加 default case

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      s := "c"
 9:
10:      switch s {
11:      case "a":
12:          fmt.Println("The letter a!")
13:      case "b":
14:          fmt.Println("The letter b!")
15:      default:
16:          fmt.Println("I don't recognize that letter!")
17:      }
18:  }
```

TRY IT YOURSELF ▼

在 switch 语句中添加 default case

在这个示例中，您将明白如何在 switch 语句中使用 default case。

1. 复制程序清单 5.6 所示的代码，这些代码也可在本书代码示例中的文件 example06.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example06.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
I don't recognize that letter!
```

5. 将变量 s 的值改为 a 并再次运行这个程序。现在会执行 default case 吗？

▲

5.8　使用 for 语句进行循环

for 语句让您能够反复执行代码块，这在编程中被称为循环：反复执行代码块，直到条件不再满足。一个这样的简单示例是，将一个数字不断加 1，直到它为特定的数字，如程序清单 5.7 所示。

程序清单 5.7 只包含条件的 for 语句

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      i := 0
 9:      for i < 10 {
10:          i++
11:          fmt.Println("i is", i)
12:      }
13:  }
```

对程序清单 5.7 解读如下。

- 声明变量 i 并将其初始化为 0。
- 一条 for 语句判断 i 是否小于 10。
- 如果这个布尔条件为 true，就执行 for 语句中的代码。
- 使用递增运算符将变量 i 加 1。
- 将变量 i 的值打印到终端。
- 返回到开头的布尔表达式，判断变量 i 的值是否小于 10。
- 变量 i 不再小于 10 时，这个布尔表达式将为 false，因此不再执行 for 语句中的代码，循环就此结束。

▼ TRY IT YOURSELF

理解 for 语句

在这个示例中，您将明白如何使用 for 语句。

1. 复制程序清单 5.7，这些代码也可在本书代码示例中的文件 example07.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example07.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
i is 1
i is 2
i is 3
i is 4
i is 5
i is 6
i is 7
i is 8
i is 9
i is 10
```

5. 注意到循环在 i 的值不再小于 10 时终止。

▲

5.8.1 包含初始化语句和后续语句的 for 语句

除要检查的条件外，for 语句还可指定在循环开始时执行的初始化语句以及后续（post）语句。这些语句可让迭代代码简短得多，但必须使用分号将它们分隔开。

- 初始化语句：仅在首次迭代前执行。
- 条件语句：每次迭代前都将检查的布尔表达式。
- 后续语句：每次迭代后都将执行。

程序清单 5.8 演示了如何在 for 语句中使用初始化语句和后续语句。

程序清单 5.8 一个包含初始化语句和后续语句的 for 语句

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      for i := 0; i < 10; i++ {
 9:          fmt.Println("i is", i)
10:      }
11:  }
```

程序清单 5.8 解读如下。

- 声明变量 i 并将其初始化为 0。
- 一条 for 语句检查变量 i 是否小于 10。
- 如果这个布尔表达式为 true，就执行 for 语句中的代码。
- 使用递增运算符将变量 i 的值加 1。
- 将变量 i 的值打印到终端。
- 返回到开头的布尔表达式，判断变量 i 的值是否小于 10。
- 变量 i 不再小于 10 时，这个布尔表达式将为 false，因此不再执行 for 语句中的代码，循环就此结束。

如果您要运行这个程序，可在本书代码示例中的文件 example08.go 中找到它。

5.8.2 包含 range 子句的 for 语句

for 语句也可用来遍历数据结构。在下面的示例中，创建了一个数组，用于存储一系列数字。数组将在第 6 章介绍，简单地说，数组用于存储一系列类型相同的数据。程序清单 5.9 演示了一条包含 range 子句的 for 语句。

程序清单 5.9 包含 range 子句的 for 语句

```
1:  package main
2:
3:  import (
4:      "fmt"
5:  )
```

```
 6:
 7:  func main() {
 8:      numbers := []int{1, 2, 3, 4}
 9:      for i, n := range numbers {
10:          fmt.Println("The index of the loop is", i)
11:          fmt.Println("The value from the array is", n)
12:      }
13:  }
```

对程序清单 5.9 解读如下。

- 声明变量 numbers，并将一个包含 4 个整数的数组赋给它。
- for 语句指定了迭代变量 i，用于存储索引值。这个变量将在每次迭代结束后更新。
- for 语句指定了迭代变量 n，用于存储来自数组中的值。它也将在每次迭代结束后更新。
- 在循环中，打印这两个变量的值。

在 Go 语言中，可使用包含 range 子句的 for 语句来遍历大多数数据结构，且无须知道数据结构的长度。

▼ TRY IT YOURSELF

使用包含 range 子句的 for 语句

在这个示例中，您将明白如何使用包含 range 子句的 for 语句。

1. 复制程序清单 5.9 所示的代码，这些代码也可在本书代码示例中的文件 example09.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example09.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
The index of the loop is 0
The value from the slice is 1
The index of the loop is 1
The value from the slice is 2
The index of the loop is 2
The value from the slice is 3
The index of the loop is 3
The value from the slice is 4
```

▲

Watch Out!

警告： 迭代变量从 0 开始。

迭代变量从 0 开始，且每次都加 1。要在特定的迭代中执行操作，务必从 0 而不是 1 开始计算迭代数！

5.9 使用 defer 语句

defer 是一个很有用的 Go 语言功能，它能够让您在函数返回前执行另一个函数。函数在

遇到 return 语句或到达函数末尾时返回。defer 语句通常用于执行清理操作或确保操作（如网络调用）完成后再执行另一个函数。程序清单 5.10 使用了一条 defer 语句，这条语句将在它所在的函数返回前执行。

程序清单 5.10　执行 defer 语句

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      defer fmt.Println("I am run after the function completes")
 9:      fmt.Println("Hello World!")
10:      }
11:  }
```

对程序清单 5.10 解读如下。

- 使用一条 defer 语句，在它所在的函数执行完毕后执行另一个函数。
- 向终端打印 Hello World!，外部函数就此结束。
- 外部函数结束后，执行 defer 语句指定的函数。

TRY IT YOURSELF ▼

理解 defer 语句

在这个示例中，您将明白如何使用 defer 语句。

1. 复制程序清单 5.10 所示的代码，这些代码也可在本书代码示例中的文件 example10.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example10.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
Hello World!
I am run after the function completes
```

▲

程序清单 5.11 演示了多条 defer 语句及其执行顺序。

程序清单 5.11　多条 defer 语句

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      defer fmt.Println("I am the first defer statement")
```


```
 9:      defer fmt.Println("I am the second defer statement")
10:      defer fmt.Println("I am the third defer statement")
11:      fmt.Println("Hello World!")
12:      }
13:  }
```

对程序清单 5.11 解读如下。

- 3 条 defer 语句都指定了它们所在的函数执行完毕后要执行的函数。
- 向终端打印 Hello World!，外部函数就此结束。
- 外部函数执行完毕后，按与 defer 语句出现顺序相反的顺序执行它们指定的函数。

▼ TRY IT YOURSELF

使用多条 defer 语句

在这个示例中，您将明白多条 defer 语句的工作原理。

1. 复制程序清单 5.11 所示的代码，这些代码也可在本书代码示例中的文件 example11.go 中找到。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example11.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
Hello World!
I am the third defer statement
I am the second defer statement
I am the first defer statement
```

▲

5.10 小结

本章介绍了如何使用控制流程来实现逻辑。您学习了如何使用 if、else if 和 else 语句来定义执行流程；然后，学习了如何使用比较运算符、算术运算符和逻辑运算符来创建布尔表达式，以定义控制流程；还学习了 if、else if 和 else 的替代品——switch 语句；接下来，您学习了 for 语句，以及如何使用只包含条件的 for 语句、包含 3 条语句的 for 语句、包含 range 子句的 for 语句来反复执行代码；最后，您学习了 defer 语句，它能够让您在外部函数返回前执行函数。至此，您具备了创建基本控制流程所需的全部知识！

5.11 问与答

问：可使用多种运算符（如比较运算符和算术运算符）来创建布尔表达式吗？

答：可以。布尔表达式可包含任意数量的运算符。例如，使用比较运算符可创建在一个数字小于 4 时返回 true 的布尔表达式，使用逻辑运算符可创建在一个数字不为 1 时返回 true

的布尔表达式。

问：该使用 else if 还是 switch 语句？

答：else if 和 switch 都是定义控制流程的合法方式。很多程序员发现，相比于使用一系列 else if，使用 switch 语句编写的代码更容易理解，因此建议使用 switch 语句。

问：为何要使用 defer 语句，而不直接使用常规的控制流程？

答：一些需要使用 defer 语句的例子包括：在读取文件后将其关闭、收到来自 Web 服务器的响应后对其进行处理以及建立连接后向数据库请求数据。需要在某项工作完成后执行特定的函数时，defer 语句是不错的选择。

5.12 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

5.12.1 小测验

1. 假设有两条 if 语句，在第一条 if 语句为 true 时，是否会忽略第二条 if 语句？
2. 请阐述逻辑运算符或是做什么的。
3. 有多条 defer 语句时，函数返回前将在按什么样的顺序执行它们？

5.12.2 答案

1. 不管第一条 if 语句的结果如何，都将评估第二条 if 语句。要在第一条 if 语句为 true 时忽略第二条 if 语句，应将其改为 else if 语句。
2. 逻辑运算符或在至少有一个表达式为 true 时返回 true。
3. 多条 defer 语句按相反的顺序执行。这意味着离函数末尾最近的 defer 语句最先执行。

5.13 练习

1. 编写一个程序，在其中声明一个整型变量，并将其设置为 88；再编写一条 if 语句，判断这个整型变量是否小于 200，如果是这样的，就向终端打印一行文本。
2. 编写一条 if 语句，用于判断一个数字是否大于 5 且小于 10。
3. 编写一条 for 语句，执行一些代码 20 次后结束。为此，您将选择 3 个 for 语句变型中的哪个？为什么？

第 6 章

数组、切片和映射

本章介绍如下内容。

- 使用数组。
- 使用切片。
- 在切片中添加和删除元素。
- 使用映射。

本章介绍 Go 语言的一些基本构件。您将学习如何声明数组以及给其元素赋值，学习 Go 语言中数组和切片的区别以及如何在切片中添加和删除元素，还将简要地了解映射。阅读本章后，您将熟悉 Go 语言编程中常用的 3 种数据结构。

6.1 使用数组

数组是一个数据集合，在编程中它通常按逻辑对数据进行分组。数组也是基本的编程构件，常用于存储一系列用数字做索引的数据。

在 Go 语言中，要创建数组，可声明一个数组变量，并指定其长度和数据类型。

```
var cheeses [2]string
```

对这些代码解读如下。

- 使用关键字 var 声明一个名为 cheeses 的变量。
- 将一个长度为 2 的数组赋给这个变量。
- 这个数组的类型为字符串。

声明变量后，便可将字符串赋给数组的元素了。

```
cheeses[0] = "Mariolles"
cheeses[1] = "Époisses de Bourgogne"
```

变量名后面的方括号和数字指定要将值赋给数组的哪个元素。索引从 0 而不是 1 开始，因此要访问数组的第一个元素，需要使用索引 0；要访问数组的第二个元素，需要使用索引 1，依此类推。

要打印数组的元素的值，可结合使用变量名和索引值。

```
fmt.Println(cheeses[0])
fmt.Println(cheeses[1])
```

另外，要打印数组的所有元素，可使用变量名本身。

```
fmt.Println(cheeses)
```

程序清单 6.1 声明了一个数组并给其赋值，再将其打印到终端。

程序清单 6.1　声明数组并给它赋值

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var cheeses [2]string
 9:      cheeses[0] = "Mariolles"
10:      cheeses[1] = "Époisses de Bourgogne"
11:      fmt.Println(cheeses[0])
12:      fmt.Println(cheeses[1])
13:      fmt.Println(cheeses)
14:  }
```


TRY IT YOURSELF ▼

理解如何声明数组并给它赋值

在这个示例中，您将明白如何声明数组、给其元素赋值以及打印其元素。

1. 打开本书代码示例中的 hour06/example01.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example01.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
Mariolles
Époisses de Bourgogne
[MariollesÉpoisses de Bourgogne]
```

▲

声明数组的长度后，就不能给它添加元素了。如果在数组 cheeses 的索引 2 处添加一个值，结果将如何呢？

```
cheeses[2] = "Camembert"
```

由于数组 cheeses 被声明为只包含两个元素，无法给第 3 个元素赋值，因此这将导致编译阶段错误。

```
./example02.go:9:  invalid array index 2 (out of bounds for 2-element array)
```

6.2 使用切片

在 Go 语言中，数组是一个重要构件，但使用切片的情况更多。切片是底层数组中的一个连续片段，通过它您可以访问该数组中一系列带编号的元素。因此，切片能够让您顺序访问数组的特定部分。为何要使用切片呢？为何不直接使用数组呢？

在 Go 语言中，使用数组存在一定的局限性。采用前面的数组 cheeses 表明方试，您无法在数组中添加元素；然而切片比数组更灵活，您可在切片中添加和删除元素，还可复制切片中的元素。可将切片视为轻量级的数组包装器，它既保留了数组的完整性，又比数组使用起来更容易。

要声明一个长度为 2 的空切片，可使用如下语法。

```
var cheeses = make([]string, 2)
```

对这些代码解读如下。

- 使用关键字 var 声明一个名为 cheeses 的变量。
- 在等号右边，使用 Go 内置函数 make 创建一个切片，其中第一个参数为数据类型，而第二个参数为长度。在这里，创建的切片包含两个字符串元素。
- 将切片赋给变量 cheeses。

创建切片后，可像给数组赋值一样给切片赋值。

```
cheeses[0] = "Mariolles"
cheeses[1] = "Époisses de Bourgogne"
```

要打印切片的值，方法也与打印数组一样。

```
fmt.Println(cheeses[0])
fmt.Println(cheeses[1])
```

到目前为止，切片类似于数组，但不同于数组的是，您可在切片中添加和删除元素。

6.2.1 在切片中添加元素

Go 语言提供了内置函数 append，让您能够增大切片的长度。

```
cheeses := append(cheeses, "Camembert")
fmt.Println(cheeses[2])
```

append 会在必要时调整切片的长度，但它对程序员隐藏了这种复杂性。在这里，将切片的长度从 2 调整为 3，并将值“Camembert”赋给了新创建的元素（其索引为 2）。在编程接口方面，程序员只需使用新创建的索引来引用这个元素即可。这样，只需一行代码，就调整了切片的长度，并给新元素赋值了。

程序清单 6.2 演示了如何在切片中添加元素。

程序清单 6.2　在切片中添加元素

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var cheeses = make([]string, 2)
 9:      cheeses[0] = "Mariolles"
10:      cheeses[1] = "Époisses de Bourgogne"
11:      cheeses = append(cheeses, "Camembert")
12:      fmt.Println(cheeses[2])
13:  }
```

TRY IT YOURSELF ▼

理解如何在切片中添加元素

在这个示例中，您将明白如何在既有切片中添加元素。

1. 打开本书代码示例中的 hour06/example03.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example03.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
Camembert
```

▲

函数 append 也是一个不定参数函数。第 4 章介绍了不定参数函数，您已经知道它们是参数数量可变的函数。这意味着使用函数 append 可在切片末尾添加很多值。

```
cheeses := append(cheeses, "Camembert", "Reblochon", "Picodon")
```

这将相应地调整切片 cheeses 的长度，并将指定的值赋给新创建的元素。

程序清单 6.3 演示了如何在切片末尾添加多个元素。

程序清单 6.3　在切片末尾添加多个元素

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var cheeses = make([]string, 2)
 9:      cheeses[0] = "Mariolles"
10:      cheeses[1] = "Époisses de Bourgogne"
11:      cheeses = append(cheeses, "Camembert", "Reblochon", "Picodon")
12:      fmt.Println(cheeses)
```

```
13: }
```

▼ TRY IT YOURSELF

在切片末尾添加多个元素

在这个示例中，您将明白如何在切片中添加多个元素。

1. 打开本书代码示例中的 hour06/example04.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example04.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
[Mariolles Époisses de Bourgogne Camembert Reblochon Picodon]
```

▲

6.2.2 从切片中删除元素

要从切片中删除元素，也可使用内置函数 append。在下面的示例中，删除了索引 2 处的元素。

```
cheeses = append(cheeses[:2], cheeses[2+1:]...)
```

通过在删除元素前后检查切片 cheeses 的长度，可知已经正确地调整了该切片的长度。另外，元素的排列顺序没有发生变化。

程序清单 6.4 演示了如何从切片中删除元素。

程序清单 6.4 从切片中删除元素

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5: )
 6:
 7: func main() {
 8:     var cheeses = make([]string, 2)
 9:     cheeses[0] = "Mariolles"
10:     cheeses[1] = "Époisses de Bourgogne"
11:     cheeses[2] = "Camembert"
12:     fmt.Println(len(cheeses))
13:     fmt.Println(cheeses)
14:     cheeses = append(cheeses[:2], cheeses[2+1:]...)
15:     fmt.Println(len(cheeses))
16:     fmt.Println(cheeses)
17: }
```

6.2.3 复制切片中的元素

要复制切片的全部或部分元素，可使用内置函数 copy。在复制切片中的元素前，必须再

声明一个类型与该切片相同的切片，例如，不能将字符串切片中的元素复制到整数切片中。程序清单 6.5 演示了如何将一个切片中的元素复制到另一个切片中。

程序清单 6.5 将一个切片的元素复制到另一个切片中

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5: )
 6:
 7: func main() {
 8:     var cheeses = make([]string, 3)
 9:     cheeses[0] = "Mariolles"
10:     cheeses[1] = "Époisses de Bourgogne"
11:     var smellyCheeses = make([]string, 2)
12:     copy(smellyCheeses, cheeses)
13:     fmt.Println(smellyCheeses)
14: }
```

函数 copy 在新切片中创建元素的副本，因此修改一个切片中的元素不会影响另一个切片。还可将单个元素或特定范围内的元素复制到新切片中，下面的示例复制索引 1 处的元素。

```
copy(smellyCheeses, cheeses[1:])
```

TRY IT YOURSELF ▼

理解如何将一个切片的元素复制到另一个切片

在这个示例中，您将明白如何将一个切片的元素复制到另一个切片。

1. 打开本书示例代码中的 hour06/example06.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example06.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
[Mariolles Époisses de Bourgogne]
```

▲

6.3 使用映射

数组和切片是可通过索引值访问的元素集合，而映射是通过键来访问的无序元素编组。大多数编程语言都支持数组；在其他编程语言中，映射也被称为关联数组、字典或散列。映射在信息查找方面的效率非常高，因为可直接通过键来检索数据。简单地说，映射可视为键-值对集合。

只需一行代码就可声明并创建一个空映射。

```
var players = make(map[string]int)
```

对这行代码解读如下。

- 关键字 var 声明一个名为 players 的变量。
- 在等号右边，使用 Go 语言内置函数 make 创建了一个映射，其键的类型为字符串，而值的类型为整数。
- 将这个空映射赋给了变量 players。

现在可在这个空映射中添加键-值对了。

```
players["cook"] = 32
players["bairstow"] = 27
players["stokes"] = 26
```

变量名后面的方括号内为键，而等号右边是要赋给键的整数值。

要打印映射中特定键对应的值，可使用这个键来获取相应的值。

```
fmt.Println(players["cook"])
fmt.Println(players["stokes"])
```

与数组和切片一样，要打印映射中所有的键-值对，可使用变量名本身。

```
fmt.Println(players)
map[cook:32 bairstow:27 stokes:26]
```

程序清单 6.6 演示了如何声明和创建映射以及如何在映射中添加元素。

程序清单 6.6 在映射中添加元素

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      var players = make(map[string]int)
 9:      players["cook"] = 32
10:      players["bairstow"] = 27
11:      players["stokes"] = 26
12:      fmt.Println(players["cook"])
13:      fmt.Println(players["bairstow"])
14:  }
```

可在映射中动态地添加元素，而无须调整映射的长度。这是 Go 语言更像 Ruby 和 Python 等动态语言，而不像 C 语言的方面之一。

从映射中删除元素

要从映射中删除元素，可使用内置函数 delete。

```
delete(players, "cook")
```

要从映射中删除键-值对，可将映射和键作为参数传递给函数 delete。在程序清单 6.7 中，从映射中删除了键为 cook 的元素。

程序清单 6.7 从映射中删除元素

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5: )
 6:
 7: func main() {
 8:     var players = make(map[string]int)
 9:     players["cook"] = 32
10:     players["bairstow"] = 27
11:     players["stokes"] = 26
12:     delete(players, "cook")
13:     fmt.Println(players)
14: }
```

TRY IT YOURSELF ▼

理解如何从映射中删除元素

在这个示例中，您将明白如何从映射中删除元素。

1. 打开本书代码示例中的 hour06/example08.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example08.go 运行这个程序。
4. 您将发现这个映射包含两个键-值对。

```
map[bairstow:27 stokes:26]
```

▲

6.4 小结

本章介绍了数组、切片和映射。您学习了如何声明数组并给其元素赋值；您知道调整数组的长度很难，而切片是便利的数组替代品；您学习了切片并知道如何在切片中添加和删除元素；最后，您了解到映射在其他语言中也被称为字典或散列，并学习了如何在映射中添加和删除由键-值对组成的元素。虽然本章重理论轻实践，但在 Go 语言中，数组、切片和映射都是基本的编程构件。

6.5 问与答

问：该使用数组还是切片？

答：除非确定必须使用数组，否则请使用切片。切片能够让您轻松地添加和删除元素，还无须处理内存分配问题。

问：没有从切片中删除元素的内置函数吗？

答：不能将 delete 用于切片。没有专门用于从切片中删除元素的函数，但可使用内置函数 append 来完成这种任务，您还可创建子切片。

问：需要指定映射的长度吗？

答：不需要。使用内置函数 make 创建映射时，可使用第二个参数，但这个参数只是容量提示，而非硬性规定。映射可根据要存储的元素数量自动增大，因此没有必要指定长度。

6.6 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

6.6.1 小测验

1. 在 Go 语言中，可调整数组的长度吗？
2. 要访问数组的第 4 个元素，应使用哪个索引号？
3. 要调整切片的长度，可使用哪个内置函数？

6.6.2 答案

1. 除非根据既有数组创建一个新数组，否则无法调整数组的长度。如果要动态地调整数组的长度，可使用切片。
2. 要访问数组的第 4 个元素，应使用索引号 3。别忘了，数组的索引从 0 开始！
3. 要在切片中添加元素，可使用内置函数 append。

6.7 练习

1. 修改本书代码示例中的 example01.go，在数组 cheeses 中包含您喜欢的奶酪。确保这个数组被声明得足够长，能够容纳您要添加的所有奶酪！
2. 创建一个包含 4 个元素的切片；再创建一个切片，并将第 3 个元素和第 4 个元素复制到该切片中。
3. 根据下述 HTML 元素列表创建一个映射，想想应将其键和值分别声明为什么类型。

- p - 段落。
- img - 图像。
- h1 - 一级标题。
- h2 - 二级标题。

第 7 章

使用结构体和指针

本章介绍如下内容。

- 结构体是什么？
- 创建结构体。
- 嵌套结构体。
- 自定义结构体数据字段的默认值。
- 比较结构体。
- 理解公有和私有值。
- 区分指针引用和值引用。

顾名思义，结构体是由数据元素组成的结构，它是一个很有用的编程构件。本章介绍结构体以及各种创建结构体的方式。您将学习结构体字段的默认值以及如何自定义这些默认值；您还将学习如何比较结构体以及导出的值；最后，您将学习指针和值在影响底层内存方面存在的微妙差异。阅读本章后，您将对结构体以及如何创建和使用它们有深入的认识。

7.1 结构体是什么

结构体是一系列具有指定数据类型的数据字段，它能够让您通过单个变量引用一系列相关的值。通过使用结构体，可在单个变量中存储众多类型不同的数据字段。存储在结构体中的值可轻松地访问和修改，这提供了一种灵活的数据结构创建方式。通过使用结构体，可提高模块化程度，还能够让您创建并传递复杂的数据结构。

您还可将结构体视为用于创建数据记录（如员工记录和机票预订）的模板。

程序清单 7.1 声明并创建了一个简单的结构体。

程序清单 7.1 声明并创建结构体

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Movie struct {
 8:      Name   string
 9:      Rating float32
11:  }
12:
13:  func main() {
14:      m := Movie{
15:          Name:  "Citizen Kane",
16:          Rating:  10,
17:      }
18:      fmt.Println(m.Name, m.Rating)
19:  }
```

对程序清单 7.1 解读如下。

- 关键字 type 指定一种新类型。
- 将新类型的名称指定为 Movie。
- 类型名右边是数据类型，这里为结构体。
- 在大括号内，使用名称和类型指定了一系列数据字段。请注意，此时没有给数据字段赋值。可将结构体视为模板。
- 在 main 函数中，使用简短变量赋值声明并初始化了变量 m，给数据字段指定的值为相应的数据类型。
- 使用点表示法访问数据字段并将其打印到控制台。

要访问结构体的数据字段，可使用点表示法：结构体变量名、圆点和要访问的数据字段的名称。

TRY IT YOURSELF

创建结构体并访问其值

在这个示例中，您将明白如何创建结构体并访问其值。

1. 打开本书代码示例中的 hour07/example01.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example01.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
Citizen Kane 10
```

7.2 创建结构体

声明结构体后，就可通过多种方式创建它。假设您已经声明了一个结构体，那么就可直接声明这种类型的变量。

```
type Movie struct {
    Name string
    Rating float32
}

var m Movie
```

这将创建一个结构体实例，并将各个数据字段设置为相应数据类型的零值。要调试或查看结构体的值，可使用 fmt 包将结构体的字段名和值打印出来。为此，可使用占位符%+v 并将其传入结构体。

```
fmt.Printf("%+v\n", m)
```

这将把字段名和值打印到终端。创建结构体时，如果没有初始化，则 Go 将把每个数据字段设置为相应数据类型的零值。

```
{Name:  Rating:0}
```

以这种方式创建结构体实例后，可使用点表示法给其字段赋值。

```
var m Movie
m.Name = "Metropolis"
m.Rating = 0.99
```

结构体数据字段的值是可变的，这意味着可动态地修改它们。例如，您可以修改电影的名称。然而，一旦结构体被声明或者实例被创建，就不能再修改其字段的数据类型了，否则将引发编译错误。在程序清单 7.2 所示的示例中，创建了一个结构体并将其赋给一个变量，然后再给这个结构体的数据字段赋值。

程序清单 7.2 声明一个类型为结构体的变量

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Movie struct {
 8:      Name   string
 9:      Rating float32
10:  }
11:
12:  func main() {
13:      var m Movie
14:      fmt.Printf("%+v\n", m)
15:      m.Name = "Metropolis"
16:      m.Rating = 0.9918:
17:      fmt.Printf("%+v\n", m)
18:  }
```

对程序清单 7.2 解读如下。

- 关键字 var 声明变量 m。
- 没有给字段赋值，因此它们默认为零值。对于字符串，零值为空字符串" "；对于 float32，零值为 0。
- 将字段的值打印到终端。
- 使用点表示法给结构体的数据字段赋值。
- 再次将结构体打印到终端，以证明其数据字段的值发生了变化。

TRY IT YOURSELF

声明一个类型为结构体的变量

在这个示例中，您将明白如何将声明类型为结构体的变量。

1. 打开本书代码示例中的 hour07/example02.go。
2. 阅读其中的代码，尝试理解它们是做什么的。如果必要，请参阅前面对程序清单 7.2 的解读。
3. 在终端中使用命令 go run example02.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
{Name:  Rating:0}
```

▲

也可使用关键字 new 来创建结构体实例，如程序清单 7.3 所示。关键字 new 创建结构体 Movie 的一个实例（为其分配内存）；将这个结构体实例赋给变量 m 后，就可像前面那样使用点表示法给数据字段赋值了。

```
m := new(Movie)
m.Name = "Metropolis"
m.Rating = 0.99
```

程序清单 7.3 使用关键字 new 创建结构体实例

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Movie struct {
 8:      Name   string
 9:      Rating float32
10:  }
11:
12:  func main() {
13:      m := new(Movie)
14:      m.Name = "Metropolis"
15:      m.Rating = 0.99
16:      fmt.Printf("%+v\n", m)
17:  }
```

还可使用简短变量赋值来创建结构体实例，此时可省略关键字 new。创建结构体实例时，可同时给字段赋值，方法是使用字段名、冒号和字段值。

```
c := Movie{Name:  "Citizen Kane", Rating:  10}
```

也可省略字段名，按字段声明顺序给它们赋值，但出于可维护性考虑，不推荐这样做。

```
c := Movie{"Citizen Kane", 10}
```

字段很多时，让每个字段独占一行能够提高代码的可维护性和可读性。请注意，如果您选择这样做，则最后一个数据字段所在的行也必须以逗号结尾。

```
c := Movie {
    Name:  "Citizen Kane",
    Rating:  10,
}
```

使用简短变量赋值是最常用的结构体创建方式，也是推荐的方式（参见程序清单 7.4）。

程序清单 7.4　使用简短变量赋值创建结构体实例

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Movie struct {
 8:      Name   string
 9:      Rating float32
10:  }
11:
12:  func main() {
13:      m := Movie{
14:          Name:  "Metropolis",
15:          Rating:  0.99,
16:      }
17:      fmt.Printf("%+v\n", m)
18:  }
```

TRY IT YOURSELF ▼

使用简短变量赋值创建结构体实例

在这个示例中，您将明白如何使用简短变量赋值创建结构体实例。

1. 打开本书代码示例中的 hour07/example05.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example05.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
{Name:Metropolis Rating:0.99}
```

▲

Did you Know?

提示： 类C语言也支持结构体。

在C和C++中，结构体很常见，作为C语言家族的一员，Go也支持结构体。结构体并非创建面向对象代码的方式，而是一种数据结构创建方式，旨在满足数据建模需求。

7.3 嵌套结构体

有时候，数据结构需要包含多个层级。此时，虽然可选择使用诸如切片等数据类型，但有时候需要的数据结构更复杂。为建立较复杂的数据结构，在一个结构体中嵌套另一个结构体的方式很有用。一个这样的例子是超级英雄列表：对于每位超级英雄，都需要存储其住址，而住址本身也是一个数据结构，非常适合使用结构体来表示。

```
type Superhero struct {
    Name    string
    Age     int
    Address Address
}

type Address struct {
    Number int
    Street string
    City   string
}
```

创建结构体Superhero的实例时，其中将包含一个数据字段为默认值的Address结构体。这可改善代码的灵活性和模块性，因为结构体Address也可用于其他地方。

可在创建Superhero结构体前创建Address结构体并给它赋值，但也可在创建Superhero结构体时这样做。程序清单7.5使用简短变量赋值创建了一个嵌套结构体实例。

程序清单7.5 使用简短变量赋值创建嵌套结构体实例

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Superhero struct {
 8:      Name    string
 9:      Age     int
10:      Address Address
11:  }
12:
13:  type Address struct {
14:      Number int
15:      Street string
16:      City   string
17:  }
18:
19:  func main() {
20:      e := Superhero{
21:          Name:  "Batman",
22:          Age:   32,
23:          Address:  Address{
24:              Number:  1007,
25:              Street:  "Mountain Drive",
26:              City:    "Gotham",
```

```
27:          },
28:      }
29:      fmt.Printf("%+v\n", m)
30:  }
```

要访问内嵌结构体的数据字段，可使用点表示法，这意味着使用结构体变量名、圆点、数据字段名、圆点和内嵌数据字段名，如下所示。

```
fmt.Println(e.Address.Street)
```

TRY IT YOURSELF ▼

使用嵌套的结构体

在这个示例中，您将明白如何使用嵌套的结构体。

1. 打开本书代码示例中的 hour07/example06.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example06.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
{Name:Batman Age:32 Address:{Number:1007 Street:Mountain Drive City:Gotham}}
```

▲

7.4 自定义结构体数据字段的默认值

创建数据结构时，自定义数据字段的默认值是很有必要的。默认情况下，Go 给数据字段指定相应数据类型的零值。表 7.1 列出了这些零值。

表 7.1 Go 语言中的零值

类型	零值
布尔型（Boolean）	false
整型（Integer）	0
浮点型（Float）	0.0
字符串（String）	""
指针（Pointer）	nil
函数（Function）	nil
接口（Interface）	nil
切片（Slice）	nil
通道（Channel）	nil
映射（Map）	nil

创建结构体时，如果没有给其数据字段指定值，它们将为表 7.1 所示的零值。Go 语言没有提供自定义默认值的内置方法，但可使用构造函数来实现这个目标。构造函数创建结构体，并将没有指定值的数据字段设置为默认值。

```
type Alarm struct {
    Time string
```

```
    Sound string
}

func NewAlarm(time string) Alarm {
    a := Alarm{
        Time:  time,
        Sound:  "Klaxon",
    }
    return a
}
```

这里不直接创建结构体 Alarm，而是使用函数 NewAlarm 来创建，从而让字段 Sound 包含自定义的默认值。请注意，这只是一种技巧，而并非 Go 语言规范的组成部分。如果您直接创建结构体 Alarm 的实例，且没有给 Sound 赋值，它将包含默认值""。通过使用构造函数来创建这种结构体的实例时，字段 Sound 的默认值将为 Klaxon。请注意，可轻松地修改字段 Sound 的值，因此这种方法创建的是初始默认值，而不是常量值。程序清单 7.6 演示了如何将字段初始化为自定义默认值。

程序清单 7.6 使用构造函数自定义默认值

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Alarm struct {
 8:      Time    string
 9:      Sound     bool
10:  }
11:
12:  func NewAlarm(time string) Alarm {
13:      a := Alarm{
14:          Time:  time,
15:          Sound: "Klaxon",
16:      }
17:      return a
18:  }
19:
20:  func main() {
21:      fmt.Printf("%+v\n", NewAlarm("07:00"))
22:  }
```

▼TRY IT YOURSELF

使用构造函数自定义默认值

在这个示例中，您将明白如何使用构造函数自定义结构体数据字段的默认值。

1. 打开本书代码示例中的 hour07/example07.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example07.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
{Time:07:00 Sound:Klaxon}
```

▲

7.5 比较结构体

对结构体进行比较，要先看它们的类型和值是否相同。对于类型相同的结构体，可使用相等性运算符来比较。要判断两个结构体是否相等，可使用==；要判断它们是否不等，可使用!=。在程序清单 7.7 中，创建了两个数据字段值相等的结构体，由于它们相等，因此比较运算符==返回 true。

程序清单 7.7 对两个数据字段值相等的结构体进行比较

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Drink struct {
 8:      Name    string
 9:      Ice     bool
10:  }
11:
12:  func main() {
13:      a := Drink{
14:          Name: "Lemonade",
15:          Ice:  true,
16:      }
17:      b := Drink{
18:          Name: "Lemonade",
19:          Ice:  true,
20:      }
21:      if a == b {
22:          fmt.Println("a and b are the same")
23:      }
24:      fmt.Printf("%+v\n", a)
25:      fmt.Printf("%+v\n", b)
26:  }
```

TRY IT YOURSELF ▼

检查两个结构体是否相等

在这个示例中，您将明白如何比较两个结构体是否相等。

1. 打开本书代码示例中的 hour07/example08.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example08.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
a and b are the same
{Name:Lemonade Ice:true}
{Name:Lemonade Ice:true}
```

▲

不能对两个类型不同的结构体进行比较，否则将导致编译错误。因此，试图比较两个结

构体之前，必须确定它们的类型相同。要检查结构体的类型，可使用 Go 语言包 reflect。在程序清单 7.8 中，使用了 reflect 包来检查结构体的类型。

程序清单 7.8 检查结构体的类型

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "reflect"
 6:  )
 7:
 8:  type Drink struct {
 9:      Name    string
10:      Ice     bool
11:  }
12:
13:  func main() {
14:      a := Drink{
15:          Name: "Lemonade",
16:          Ice:  true,
17:      }
18:      b := Drink{
19:          Name: "Lemonade",
20:          Ice:  true,
21:      }
22:      fmt.Printf(reflect.TypeOf(a))
23:      fmt.Printf(reflect.TypeOf(b))
24:  }
```

▼ TRY IT YOURSELF

检查结构体的类型

在这个示例中，您将检查结构体的类型。

1. 打开本书代码示例中的 hour07/example09.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example09.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
main.Drink
main.Drink
```

▲

7.6 理解公有和私有值

如果您熟悉其他编程语言，可能明白公有和私有数据的概念。如果一个值被导出，可在函数、方法或包外面使用，那么它就是公有的；如果一个值只能在其所属上下文中使用，那么它就是私有的。第 13 章将更详细地介绍包和导出。

根据 Go 语言约定，结构体及其数据字段都可能被导出，也可能不导出。如果一个标识符的首字母是大写的，那么它将被导出；否则不会导出。

要导出结构体及其字段，结构体及其字段的名称都必须以大写字母打头。

7.7 区分指针引用和值引用

使用结构体时，明确指针引用和值引用的区别很重要。在介绍如何给结构体添加方法的第 8 章中，这一点也很重要。

第 3 章说过，数据值存储在计算机内存中。指针包含值的内存地址，这意味着使用指针可读写存储的值。创建结构体实例时，给数据字段分配内存并给它们指定默认值；然后返回指向内存的指针，并将其赋给一个变量。使用简短变量赋值时，将分配内存并指定默认值。

```
a := Drink{}
```

复制结构体时，明确内存方面的差别很重要。将指向结构体的变量赋给另一个变量时，被称为赋值。

```
a := b
```

赋值后，a 与 b 相同，但它是 b 的副本，而不是指向 b 的引用。修改 b 不会影响 a，反之亦然。程序清单 7.9 演示了这种行为。

程序清单 7.9 以值引用的方式复制结构体

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5:     "reflect"
 6: )
 7:
 8: type Drink struct {
 9:     Name    string
10:     Ice     bool
11: }
12:
13: func main() {
14:     a := Drink{
15:         Name: "Lemonade",
16:         Ice:  true,
17:     }
18:     b := a
19:     b.Ice = false
20:     fmt.Printf("%+v\n", b)
21:     fmt.Printf("%+v\n", a)
22:     fmt.Printf("%p\n", &a)
23:     fmt.Printf("%p\n", &b)
24: }
```

对程序清单 7.9 解读如下。

- 声明结构体类型 Drink。
- 创建结构体 Drink 的一个实例，并将其赋给变量 a。

- 声明变量 b 并将 a 赋给它。
- 修改 b 的数据字段 Ice。
- 将 b 的值打印到终端。
- 将 a 的值打印到终端，以证明修改 b 不会影响 a。
- 使用 fmt.Printf 将 a 和 b 的内存地址打印到终端，以证明它们的内存地址不同。

TRY IT YOURSELF

以值引用的方式复制结构体

在这个示例中，您将明白如何以值引用的方式复制结构体。

1. 打开本书代码示例中的 hour07/example10.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example10.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
{Name:Lemonade Ice:false}
{Name:Lemonade Ice:true}
0xc42000a1e0
0xc42000a200
```

要修改原始结构体实例包含的值，必须使用指针。指针是指向内存地址的引用，因此使用它操作的不是结构体的副本而是其本身。要获得指针，可在变量名前加上和号。可对程序清单 7.9 进行修改，以使用指针引用而不是值引用，如程序清单 7.10 所示。

程序清单 7.10 以指针引用的方式复制结构体

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5:     "reflect"
 6: )
 7:
 8: type Drink struct {
 9:     Name    string
10:     Ice     bool
11: }
12:
13: func main() {
14:     a := Drink{
15:         Name:  "Lemonade",
16:         Ice:   true,
17:     }
18:     b := &a
19:     b.Ice = false
20:     fmt.Printf("%+v\n", *b)
21:     fmt.Printf("%+v\n", a)
22:     fmt.Printf("%p\n", b)
```

```
23:       fmt.Printf("%p\n", &a)
24:   }
```

相比于程序清单 7.9，程序清单 7.10 的不同之处如下。

- 将指向 a 的指针（而不是 a 本身）赋给 b，这是使用和号字符表示的。
- 修改 b 时，将修改分配给 a 的内存，因为 a 和 b 指向相同的内存。
- 打印 a 和 b 的值时，将发现它们的值相同。请注意，由于 b 是指针，因此必须使用星号字符对其进行引用。
- 将 b 和 a 的内存地址打印到控制台，以证明它们相同。

TRY IT YOURSELF ▼

以指针引用的方式复制结构体

在这个示例中，您将明白如何以指针引用的方式复制结构体。

1. 打开本书代码示例中的 hour07/example11.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example11.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
{Name:Lemonade Ice:false}
{Name:Lemonade Ice:false}
0xc42000a1e0
0xc42000a1e0
```

▲

指针和值的差别很微妙，但选择使用指针还是值很容易区分：如果需要修改原始结构体实例，就使用指针；如果要操作一个结构体，但不想修改原始结构体实例，就使用值。

7.8 小结

本章介绍了结构体。您了解到结构体非常适合用来表示电影、超级英雄和饮料等；您知道如何创建结构体以及创建结构体时将给它们指定默认值；您明白了如何在创建结构体时给它们指定自定义默认值；您学习了导出的值以及指针和值之间的差别。现在，您能够将周遭的事物视为结构体，例如，对于小鸟、星星和运动员，您将如何使用结构体来表示它们呢？结构体是一个功能强大的基本构件，提供了一种编程思维。

7.9 问与答

问：本书前面介绍了 3 种创建结构体的方式，我该使用哪种呢？

答：推荐使用简短变量赋值方式（:=）来创建结构体。关键字 new 和变量声明方式也合法，但不那么常用。

问：我明白了如何嵌套结构体，请问最多可嵌套多少层呢？

答：对结构体嵌套层级数没有任何限制，但如果嵌套层级太多，可能昭示着使用其他数据结构是更好的选择。

问：结构体数据字段可以是任何数据类型吗？

答：是的，在结构体中可使用任何数据类型，包括自定义类型。

7.10 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

7.10.1 小测验

1. 结构体数据字段默认为什么值？
2. 如何指定是否要将结构体及其数据字段导出？
3. 什么情况下该使用指针？什么情况下该使用值？

7.10.2 答案

1. 结构体数据字段默认为零值：根据字段的数据类型，将其设置为相应的零值。完整的零值列表请参阅表 7.1。

2. 以大写字母打头的标识符将被导出，以小写字母打头的标识符不会被导出。例如，标识符 Age 会被导出，而 age 不会。

3. 要创建结构体的副本，但不希望修改影响原始结构体时，应使用值；要操作结构体的副本，并希望所做的修改在原始结构体中反映出来时，应使用指针。

7.11 练习

1. 声明一个结构体，用于表示您很喜欢玩的控制台游戏。请考虑游戏名称、发布日期和适用平台等信息，并选择合适的数据类型。在 main 函数中，创建该结构体的一个实例，并将其数据字段打印到终端。

2. 想出一种可使用嵌套结构体来表示的数据结构。一些这样的例子包括：大型建筑物的组成部分、简历中的条目、房子中的起居室。

3. 阐述传递指针和传递值的差别，这种差别在内存方面意味着什么？

第 8 章

创建方法和接口

本章介绍如下内容。

- 使用方法。
- 创建方法集。
- 使用方法和指针。
- 使用接口。

第 7 章介绍了结构体，您明白了它是一种创建数据结构的方式，还知道可使用点表示法来访问结构体中的数据。然而，涉及更复杂的操作时，理解和处理起来就不那么容易了。Go 提供了另一种操作数据的方式——通过方法来操作。本章首先介绍方法以及如何创建和使用与特定数据类型相关联的方法集，再介绍一种描述方法集的方式——接口。

8.1 使用方法

方法类似于函数，但有一点不同：在关键字 func 后面添加了另一个参数部分，用于接受单个参数。下面的示例给第 7 章介绍的结构体 Movie 添加了一个方法。

```
type Movie struct {
    Name string
    Rating float32
}

func (m *Movie) summary() string {
    //code
}
```

请注意，在方法声明中，关键字 func 后面多了一个参数——接收者。严格地说，方法接收者是一种类型，这里是指向结构体 Movie 的指针。接下来是方法名、参数以及返回类型。除多了包含接收者的参数部分外，方法与第 4 章介绍的函数完全相同。可将接收者视为与方

法相关联的东西。通过声明方法 summary，让结构体 Movie 的任何实例都可使用它。为何要使用方法，而不直接使用函数呢？例如，下面的函数与前面的方法声明等价。

```
type Movie struct {
    Name string
    Rating float64
}

func summary(m *Movie) string {
    //code
}
```

函数 summary 和结构体 Movie 相互依赖，但它们之间没有直接关系。例如，如果不能访问结构体 Movie 的定义，就无法声明函数 summary。如果使用函数，则在每个使用函数或结构体的地方，都需包含函数和结构体的定义，这会导致代码重复。另外，函数发生任何改变，都必须随之修改多个地方。这样看来在函数与结构体关系密切时，使用方法更合理。

方法 summary 的实现将 float64 等级制转换为字符串并设置其格式。使用方法的优点在于，只需编写方法实现一次，就可对结构体的任何实例进行调用。

```
func (m *Movie) summary() string {
    r := strconv.FormatFloat(m.Rating, 'f', 1, 64)
    return m.Name + ", " + r
}
```

程序清单 8.1 声明了方法 summary，并对 Moive 结构体的一个实例进行调用。

程序清单 8.1　声明并调用方法

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "strconv"
 6:  )
 7:
 8:  type Movie struct {
 9:      Name   string
10:      Rating float64
11:  }
12:
13:  func (m *Movie) summary() string {
14:      r := strconv.FormatFloat(m.Rating, 'f', 1, 64)
15:      return m.Name + ", " + r
16:  }
17:
18:  func main() {
19:      m := Movie{
20:      Name:  "Spiderman",
21:          Rating:  3.2,
22:      }
23:
24:      fmt.Println(m.summary())
25:  }
```

TRY IT YOURSELF ▼

声明并调用方法

在这个示例中，您将明白如何给结构体声明方法并调用它。

1. 打开本书代码示例中的 hour08/example01.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example01.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
Spiderman, 3.2
```

▲

8.2 创建方法集

方法集是可对特定数据类型进行调用的一组方法。在 Go 语言中，任何数据类型都可有相关联的方法集，这让您能够在数据类型和方法之间建立关系，如前面的结构体 Movie 示例所示。方法集可包含的方法数量不受限制，这是一种封装功能和创建库代码的有效方式。

处理球体时，假设您要计算其表面积和体积。在这种情况下，非常适合使用结构体和方法集。通过使用方法集，您只需创建一次计算代码，就可将其重用于任何球体。要创建这个方法集，可声明结构体 Sphere，再声明两个将结构体 Sphere 作为接收者的方法。

```
type Sphere struct {
    Radius float64
}

func (s *Sphere) SurfaceArea() float64 {
    return float64(4) * math.Pi * (s.Radius * s.Radius)
}

func (s *Sphere) Volume() float64 {
    radiusCubed := s.Radius * s.Radius * s.Radius
    return (float64(4) / float64(3)) * math.Pi * radiusCubed
}
```

这里声明了计算球体表面积和体积的方法，并像通常那样定义函数签名。唯一不同的是添加了一个表示接收者的参数，这里是一个指向 Sphere 实例的指针。就本章而言，方法使用的公式并不重要，它们都是标准的数学公式。然而，需要指出的是，在方法中可以访问结构体的 Radius 值，这是使用点表示法访问的。程序清单 8.2 定义了一个与结构体相关联的方法集。

程序清单 8.2 创建并使用方法集

```
1:  package main
2:
3:  import (
```

```
 4:      "fmt"
 5:      "math"
 6:  )
 7:
 8:  type Sphere struct {
 9:      Radius float64
10:  }
11:
12:  func (s *Sphere) SurfaceArea() float64 {
13:      return float64(4) * math.Pi * (s.Radius * s.Radius)
14:  }
15:
16:  func (s *Sphere) Volume() float64 {
17:      radiusCubed := s.Radius * s.Radius * s.Radius
18:      return (float64(4) / float64(3)) * math.Pi * radiusCubed
19:  }
20:
21:  func main() {
22:
23:      s := Sphere{
24:          Radius:  5,
25:      }
26:      fmt.Println(s.SurfaceArea())
27:      fmt.Println(s.Volume())
28:  }
```

相比于使用函数，使用方法集的优点在于，只需编写一次方法 SurfaceArea 和 Volume。例如，如果发现这两个方法中有一个存在 Bug，则只需修改一个地方即可。

▼ TRY IT YOURSELF

创建并使用方法集

在这个示例中，您将明白如何创建并使用方法集。

1. 打开本书代码示例中的 hour08/example02.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example02.go 运行这个程序。
4. 您将在终端中看到如下文本。

```
314.1592653589793
523.5987755982989
```

▲

8.3 使用方法和指针

与结构体一样，明白如何使用方法和指针也很重要。您已经看到，方法是一个接受被称为接收者的特殊参数的函数，接收者可以是指针，也可以是值，但两者的差别非常微妙。假设有一个存储三角形数据的结构体。

```
type Triangle struct {
```

```
    width float64
    height float64
}
```

为计算三角形的面积，一个简单的公式是将高度和底相乘，再乘以 1/2。虽然这可以直接使用结构体的数据字段来计算，但更简结的方式是定义一个方法。方法 area 返回前述公式的结果。请注意，接收者是指向结构体 Triangle 的指针，这是由星号指定的。

```
func (t *Triangle) area() float64 {
    return 0.5 * (t.width * t.height)
}
```

程序清单 8.3 演示了如何向方法传递指针引用。

程序清单 8.3　向方法传递指针引用

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Triangle struct {
 8:      base float64
 9:      height float64
10:  }
11:
12:  func (t *Triangle) area() float64 {
13:      return 0.5 * (t.base * t.height)
14:  }
15:
16:  func main() {
17:      t := Triangle{base:  3, height:  1}
18:      fmt.Println(t.area())
19:  }
```

与您期望的一样，执行这个示例将返回三角形的面积。

```
go run example03.go
1.5
```

为理解将接收者参数声明为指针引用和值引用的差别，我们来看一个简单的示例，它修改结构体中定义的三角形的底值。假设要修改三角形的底值，可使用方法 changeBase 来实现。

```
func (t Triangle) changeBase(f float64) {
    t.base = f
    return
}
```

在这个示例中，注意到指定接收者参数类型时没有在 Triangle 前面加上星号，这意味着接收者参数是值而不是指针。程序清单 8.4 是一个完整的示例，它在结构体 Triangle 的方法集中添加了方法 changeBase。

程序清单 8.4　向方法传递值引用

```
 1:  package main
 2:
```

```
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Triangle struct {
 8:      base float64
 9:      height float64
10:  }
11:
12:  func (t Triangle) changeBase(f float64) {
13:      t.base = f
14:      return
15:  }
16:
17:  func main() {
18:      t := Triangle{base:  3, height:  1}
19:      t.changeBase(4)
20:      fmt.Println(t.base)
21:  }
```

您认为这个程序打印的 t.base 的值会是多少呢？

▼ TRY IT YOURSELF

向方法传递值引用

在这个示例中，您将明白向方法传递值引用意味着什么。

1. 打开本书代码示例中的 hour08/example04.go。
2. 阅读其中的代码，尝试理解它们是做什么的。请注意，向方法传递的是值引用。
3. 在终端中使用命令 go run example04.go 运行这个程序。
4. 您将在终端中看到如下文本，这符合您的预期吗？

```
3
```

▲

之所以打印的是 3，是因为方法 changeBase 接受的是一个值引用。这意味着这个方法操作的是结构体 Triangle 的副本，而原始结构体不受影响。在方法 changeBase 中，修改的是原始 Triangle 结构体的副本的 t.base。

将指针作为接收者的方法能够修改原始结构体的数据字段，这是因为它使用的是指向原始结构体内存单元的指针，因此操作的不是原始结构体的副本。

可修改程序清单 8.5，将接收者参数的类型声明为指针，如程序清单 8.5 所示。这样调用方法时，将修改原始结构体的数据字段。

程序清单 8.5 向方法传递指针引用

```
1:  package main
2:
3:  import (
4:      "fmt"
5:  )
6:
```

```
 7:  type Triangle struct {
 8:      base float64
 9:      height float64
10:  }
11:
12:  func (t *Triangle) changeBase(f float64) {
13:      t.base = f
14:      return
15:  }
16:
17:  func main() {
18:      t := Triangle{base:  3, height:  1}
19:      t.changeBase(4)
20:      fmt.Println(t.base)
21:  }
```

您认为这个程序打印的 t.base 值会是多少呢？

TRY IT YOURSELF ▼

向方法传递指针引用

在这个示例中，您将明白向方法传递指针引用意味着什么。

1. 打开本书代码示例中的 hour08/example05.go。
2. 阅读其中的代码，尝试理解它们是做什么的。请注意，向方法传递的是指针引用。
3. 在终端中使用命令 go run example05.go 运行这个程序。
4. 您将在终端中看到如下文本，这符合您的预期吗？

```
4
```

▲

指针和值之间的差别很微妙，但选择使用指针还是值这一点很简单：如果需要修改原始结构体，就使用指针；如果需要操作结构体，但不想修改原始结构体，就使用值。

8.4 使用接口

在 Go 语言中，接口指定了一个方法集，这是实现模块化的强大方式。您可将接口视为方法集的蓝本，它描述了方法集中的所有方法，但没有实现它们。接口功能强大，因为它充当了方法集规范，这意味着可在符合接口要求的前提下随便更换实现。

接口描述了方法集中的所有方法，并指定了每个方法的函数签名。下面的示例假设需要编写一些控制机器人（Robot）的代码。粗略地说，可假定有多种类型的机器人，控制这些机器人行为的方式存在细微的差别。给定这个编程任务，您可能认为需要为每种机器人编写不同的代码。通过使用接口，可将代码重用于有相同行为的实体。就这个机器人示例而言，下面的接口描述了开关机器人的方式。

```
type Robot interface {
    PowerOn() err
```

```
}
```

接口 Robot 只包含一个方法——PowerOn。这个接口描述了方法 PowerOn 的函数签名：不接受任何参数且返回一种错误类型。从高级层面说，接口还有助于理解代码设计。在无须关心实现的情况下，很容易理解设计是什么样的。

那么如何使用接口呢？接口是方法集的蓝本，要使用接口，必须先实现它。如果代码满足了接口的要求，就实现了接口。要实现接口 Robot，可声明一个满足其要求的方法集。

```
type T850 struct {
    Name string
}

func (a *T850) PowerOn() err {
    return nil
}
```

这个实现很简单，但满足了接口 Robot 的要求，因为它包含方法 PowerOn，且这个方法的函数签名与接口 Robot 要求的一致。接口的强大之处在于，它们支持多种实现。例如，您也可以像下面这样来实现接口 Robot。

```
type R2D2 struct {
    Broken bool
}

func (r *R2D2) PowerOn() err {
    if r.Broken {
        return errors.New("R2D2 is broken")
    } else {
        return nil
    }
}
```

这也满足了接口 Robot 的要求，因为它符合这个方法集的定义——包含方法 PowerOn，同时函数签名也相同。请注意，这里与方法集相关联的结构体为 R2D2，它包含的数据字段与 T850 不同，方法 PowerOn 的代码也完全不同，但函数签名一样。

要满足接口的要求，只要实现了它指定的方法集，且函数签名正确无误就可以了。

当前，接口 Robot 有两种实现，虽然有相同的 Robot 定义很有用，但没有可同时用于 T850 和 R2D2 实例的代码。接口也是一种类型，可作为参数传递给函数，因此可编写可重用于多个接口实现的函数。

例如，编写一个可用于启动任何机器人的函数。

```
func Boot(r Robot) error {
    return r.PowerOn()
}
```

这个函数将接口 Robot 的实现作为参数，并返回调用方法 PowerOn 的结果。这个函数可用于启动任何机器人，而不管其方法 PowerOn 是如何实现的。T850 和 R2D2 都可利用这个方法。

程序清单 8.6 是一个完整的使用接口 Robot 的示例。

程序清单 8.6 使用接口 Robot

```
 1: package main
 2:
 3: import (
 4:     "errors"
 4:     "fmt"
 5: )
 6:
 7: type Robot interface {
 8:     PowerOn() error
 9: }
10:
11: type T850 struct {
12:     Name string
13: }
14:
15: func (a *T850) PowerOn() error {
16:     return nil
17: }
18:
19: type R2D2 struct {
20:     Broken bool
21: }
22:
23: func (r *R2D2) PowerOn() error {
24:     if r.Broken {
25:         return errors.New("R2D2 is broken")
26:     } else {
27:         return nil
28:     }
29: }
30:
31: func Boot(r Robot) error {
32:   return r.PowerOn()
33: }
34:
35: func main() {
36:     t := T850{
37:         Name:  "The Terminator",
38:     }
39:
40:     r := R2D2{
41:         Broken:  true,
42:     }
43:
44:     err := Boot(&r)
45:
46:     if err != nil {
47:         fmt.Println(err)
48:     } else {
49:         fmt.Println("Robot is powered on!")
50:     }
51:
52:     err = Boot(&t)
53:
54:     if err != nil {
55:         fmt.Println(err)
56:     } else {
57:         fmt.Println("Robot is powered on!")
58:     }
59: }
```

By the Way

注意： Go 语言是面向对象的吗？

对结构体和方法有基本认识后，如果您熟悉其他语言，可能会问：Go 是面向对象的吗？面向对象编程是这样一种编程范式：使用具有特定行为的对象来建立数据模型。通常，面向对象语言提供允许一种对象继承另一种对象的功能。虽然 Go 语言没有提供类和类继承等面向对象功能，但结构体和方法集弥补了这部分不足，提供了一些面向对象元素。由此可以说 Go 在不使用类和继承的情况下提供了类似于面向对象编程的功能，虽然这一点还存在争议。

乍一看，接口提供的抽象层可能有点复杂，但它有助于代码重用，还能够让您完全更换实现。假设要编写一个使用 MySQL 数据库的计算机程序，如果不使用接口，则代码将完全是针对 MySQL 的。在这种情况下，如果后来要将 MySQL 数据库换成其他数据库，如 PostgreSQL，就可能需要重写大量的代码。

通过定义一个数据库接口，该接口的实现将比使用的数据库更重要。从理论上说，只要实现满足接口的要求，就可使用任何数据库，因此可轻松地更换数据库。数据库接口可包含多个实现，这就引入了多态的概念。

多态意味着多种形式，它让接口能够有多种实现。在 Go 语言中，接口以声明的方式提供了多态，因为接口描述了必须提供的方法集以及这些方法的函数签名。如果一个方法集实现了一个接口，就可以说它与另一个实现了该接口的方法集互为多态。编译器也会验证接口：检查方法集并确保接口确实是多态的。通过将接口正式化，可确保接口的两种实现是多态的。这无疑会让代码可验证、可测试且是灵活的。

8.5 小结

本章介绍了方法和接口，您现在应该知道如何声明方法，以及与使用函数相比，方法有助于提高代码的可重用性和一致性；您了解到方法集是可对特定数据类型执行的操作；然后学习了值引用和指针引用的差别，还有操作原始结构体和操作其副本之间的差别；最后，您大致了解了接口，它是一种创建多态实现的强大方式，这些实现可共享方法或完全更换。

8.6 问与答

问：函数和方法有何不同？

答：严格地说，方法和函数的唯一差别在于，方法多了一个指定接收者的参数，这让您能够对数据类型调用方法，从而提高代码重用性和模块化程度。

问：在什么情况下应使用指针引用？在什么情况下应使用值引用？

答：如果需要修改原始结构体中的数据，就使用指针；如果要操作原始数据的副本，就使用值引用。

问：接口的实现可包含接口中没有的方法吗？

答：可以。可在接口的实现中添加额外的方法，但这仅适用于结构体，而不适用于接口。

8.7 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

8.7.1 小测验

1. 使用方法有何优点？
2. 可将方法与切片相关联吗？
3. 多态是什么意思？

8.7.2 答案

1. 使用方法意味着只需编写一次代码。使用方法可改善易用性、一致性和可靠性。
2. 可以。在 Go 语言中，可将方法集与任何类型相关联。
3. 多态指的是多种不同的形式。在 Go 语言中，接口支持多种实现，它能够让您共享或更换代码。

8.8 练习

1. 为出租车编写一个接口。您可在其中包含任何方法，但需要考虑一些因素，如出租车是否是空的、有多少位乘客以及出租车是否停运了。
2. 扩展接口 Robot，在其中添加方法 Talk。再修改程序清单 8.6，让结构体 T850 和 R2D2 都实现方法 talk。
3. 阅读 errors 包的源代码，您明白了方法是如何与结构体相关联的吗？

第9章

使用字符串

本章介绍如下内容。

- 创建字符串字面量。
- 理解 **rune** 字面量。
- 拼接字符串。
- 编码。

字符串是基本的编程构件，本章介绍如何在 Go 语言中使用字符串，您将明白如何创建字符串以及 Go 语言中的 rune 概念，还将学习如何操作字符串并探索字符串背后的编码系统。

9.1 创建字符串字面量

Go 语言支持两种创建字符串字面量的方式。解释型字符串字面量是用双引号括起的字符，如"hello"。一种创建字符串的简单方式是使用解释型字符串字面量，如程序清单 9.1 所示。

程序清单 9.1 创建字符串字面量

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      s := "I am an interpreted string literal"
 9:      fmt.Println(s)
10:  }
```

TRY IT YOURSELF ▼

创建解释型字符串字面量

在这个示例中，您将明白如何创建解释型字符串字面量。

1. 在文本编辑器中打开文件 hour09/example01.go，并尝试理解这个示例是做什么的。
2. 在终端中使用命令 go run example01.go 运行这个程序。
3. 您将在控制台中看到打印出来的字符串字面量。

```
I am an interpreted string literal
```

▲

除换行符和未转义的双引号外，解释型字符串字面量可包含其他任何字符。对于前面有反斜杠（\）的字符，将像它们出现在 rune 字面量中那样进行解读。

表 9.1 摘自《Go 语言规范》，详细说明了单字符转义序列对应的 Unicode 字符。

表 9.1　　特殊的英语字符

rune 字面量	Unicode 字符
\a	U+0007 alert or bell
\b	U+0008（退格）
\f	U+000C（换页符）
\n	U+000A（换行符）
\r	U+000D（回车）
\t	U+0009（水平制表符）
\v	U+000b（垂直制表符）
\\	U+005c（反斜杠）
\'	U+0027（单引号，这个转义序列只能包含在 rune 字面量中）
\\"	U+0022（双引号，这个转义序列只能包含在字符串字面量中）

9.2 理解 rune 字面量

通过使用 rune 字面量，可将解释型字符串字面量分成多行，还可在其中包含制表符和其他格式选项。在程序清单 9.2 中，使用 rune 字面量添加换行符和制表符，虽然字符串声明位于一行中。

程序清单 9.2　使用 rune 字面量

```
1:  package main
2:
3:  import (
4:      "fmt"
5:  )
6:
```

```
 7:  func main() {
 8:        s := "After a backslash, certain single character escapes represent
    special values\nn is a line feed or new line \n\t t is a tab"
 9:        fmt.Println(s)
10:  }
```

运行这个示例将生成如下经过格式设置的字符串。

```
go run example02.go
After a backslash, certain single character escapes represent special values
n is a line feed or new line
t is a tab
```

原始字符串字面量用反引号括起，如'hello'。不同于解释型字符串，原始字符串中的反斜杠没有特殊含义，Go 按原样解释这种字符串。通过使用原始字符串字面量，无须利用反斜杠就能生成前一个示例那样的输出，如程序清单 9.3 所示。

程序清单 9.3 原始字符串字面量

```
 1:  package main
 2:
 3:  import (
 4:        "fmt"
 6:  )
 7:
 8:  func main() {
 9:        s := `After a backslash, certain single character escapes represent
    special values
10:  n is a line feed or new line
11:       t is a tab`
12:         fmt.Println(s)
13:  }
```

▼TRY IT YOURSELF

使用原始字符串字面量

在这个示例中，您将明白如何创建原始字符串字面量。

1. 在文本编辑器中打开文件 hour09/example03.go，并尝试理解这个示例是做什么的。
2. 在终端中使用命令 go run example03.go 运行这个程序。
3. 您将在控制台中看到打印出来的原始字符串字面量。
4. 编辑这个文件，在原始字符串中添加一些空格和其他格式选项。您明白解释型字符串字面量和原始字符串字面量的不同了吗？

▲

9.3 拼接字符串

在 Go 语言中，要拼接（合并）字符串，可将运算符+用于字符串变量。字符串是使用解释型字符串字面量还是原始字符串字面量创建的无关紧要。运算符+将它左边和右边的字符

串合并成一个字符串，如程序清单 9.4 所示。

程序清单 9.4　将多个字符串拼接成一个

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      s := "Oh sweet ignition" + " be my fuse"
 9:      fmt.Println(s)
10:  }
```

还可使用复合赋值运算符+=来拼接字符串，如程序清单 9.5 所示。这个运算符将它右边的字符串合并到左边的字符串中。例如，可在循环中反复这样做来生成字符串。

程序清单 9.5　使用复合赋值运算符拼接字符串

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      s := "Can you hear me?"
 9:      s += "\nHear me screamin'?"
10:  }
```

TRY IT YOURSELF ▼

使用复合赋值运算符拼接字符串

这个示例演示了如何使用复合赋值运算符来拼接字符串。

1. 在文本编辑器中打开文件 hour09/example05.go，尝试理解这个示例是做什么的。
2. 在终端中使用命令 go run example05.go 运行这个程序。
3. 您将在控制台中看到拼接得到的字符串。

▲

只能拼接类型为字符串的变量，如果将整数和字符串进行拼接将导致编译错误，如程序清单 9.6 所示。

程序清单 9.6　拼接和类型

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 7:  )
 8:
 9:  func main() {
10:      var i int = 1
```

```
11:       var s string = " egg"
12:       var breakfast string = i + s
13:       fmt.Println(breakfast)
14:   }
```

对程序清单 9.6 解读如下。

1. 声明整数变量 i 并将其初始化为 1。
2. 声明字符串变量 s 并将其初始化为 egg。
3. 声明字符串变量 breakfast 并将拼接 i 和 s 的结果赋给它。
4. 打印变量 breakfast 的值。

您认为这些代码是做什么的呢？运行这些代码将导致编译错误。

```
go run types.go
# command-line-arguments
./types.go:8:  invalid operation:  i + s (mismatched types int and string)
```

错误消息指出不能将整数和字符串进行拼接。那么如何拼接这些值呢？Go 标准库提供了 strconv 包，您可使用其中的方法 Itoa 来完成这种任务：它将整数转换为字符串，如程序清单 9.7 所示。

程序清单 9.7 将其它类型转换为字符串

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 6:  )
 7:
 8:  func main() {
 9:      var i int = 1
10:      var s string = " egg"
11:      intToString := strconv.Itoa(i)
12:      var breakfast string = intToString + s
13:      fmt.Println(breakfast)
14:  }
```

运行这个示例将输出拼接得到的字符串。

```
go run strconv.go
1 egg
```

9.3.1 使用缓冲区拼接字符串

对于简单而少量的拼接，使用运算符+和+=的效果虽然很好，但随着拼接操作次数的增加，这种做法的效率并不高。如果需要在循环中拼接字符串，则使用空的字节缓冲区来拼接的效率更高。在程序清单 9.8 中，使用了一个运行 500 次的循环来生成字符串。这虽然可使用运算符+=来完成，但使用缓冲区的速度要快得多。

程序清单 9.8 使用缓冲区拼接字符串

```
 1:  package main
 2:
 3:  import (
```

```
    "fmt"
)

func main() {
    var buffer bytes.Buffer

    for i := 0; i < 500; i++ {
        buffer.WriteString("z")
    }

    fmt.Println(buffer.String())
}
```

对程序清单 9.8 解读如下。

1. 创建一个空的字节缓冲区，并将其赋给变量 buffer。
2. 一个运行 500 次的循环，每次循环都将字符串"z"写入缓冲区。
3. 循环结束后，对缓冲区调用函数 String( )以字符串的方式输出结果。

TRY IT YOURSELF ▼

使用字节缓冲区来创建字符串

在这个示例中，您将明白如何使用字节缓冲区来创建字符串。

1. 在文本编辑器中打开文件 hour09/example08.go，尝试理解这个示例是做什么的。
2. 在终端中使用命令 go run example08.go 运行这个程序。
3. 您将在控制台中看到一个由 z 组成的很长的字符串。

▲

9.3.2 理解字符串是什么

要理解字符串是什么，必须明白计算机是如何显示和存储字符的。计算机将数据解读为数字，另外，虽然您根本看不到，但计算机实际上是将字符存储为数字的。

历史上有很多编码标准，最后大家就如何将字符映射到数字达成了一致。ASCII（美国信息交换标准码）曾经是最重要的编码标准，它就如何使用数字来表示英语字母表中的字符进行了标准化。

ASCII 编码标准定义了如何使用 7 位的整数（通俗地说是数字）来表示 128 个字符。表 9.2 列出了 ASCII 字符集中的一些字符，如果您看不懂，也不用担心，只要知道数字映射到字符就可以了。

表 9.2 ASCII 字符集中的一些字符

二进制	八进制	十进制	十六进制	字符
1000001	101	65	41	A
1000010	102	66	42	B
1110100	164	116	74	t

虽然 ASCII 在英语字符标准化的道路上迈出了重要的一步，但它不包含其他任何语言的

字符集。简而言之，它支持使用英语说“hello”，但不支持使用中文说“您好”。

鉴于此，Unicode 编码方案于 1987 年应运而生，它支持全球各地的大多数字符集。最新的版本支持 128000 个字符，涵盖 135 种或现代或古老的语言。更重要的是，Unicode 涵盖了 ASCII 标准，其开头的 128 个字符就是 ASCII 字符。

很多字符编码方案都实现了 Unicode 标准，其中最著名的是 UTF-8。更巧的是，Go 语言的两位设计者 Rob Pike 和 Ken Thompson 也是 UTF-8 的联合设计者。可见，Go 语言支持 UTF-8 和国际字符集，而 Go 源代码也是 UTF-8 的。

要更深入地理解字符串以及如何操作它们，必须首先知道 Go 语言中的字符串实际上是只读的字节切片。要获悉字符串包含多少个字节，可使用 Go 语言的内置函数 len。

```
s := "hello"
fmt.Printf(len(s))
// outputs 5
```

西语字符（如 a、b、c）通常映射到单个字节。单词 hello 为 5 个字节，由于 Go 字符串为字节切片，因此可输出字符串中特定位置的字节值。在下面的示例中，输出了字符串“hello”的第一个字节。

```
s := "hello"
fmt.Printf(s[0])
//outputs 104
```

您可能认为结果应为字符 h，但由于通过索引访问字符串时，访问的是字节而不是字符，因此显示的是以十进制表示的字节值。表 9.3 列出了多个字母的十进制和二进制表示。

表 9.3 字符的二进制和十进制表示

	h	e	l	l	o
十进制	104	101	108	108	111
二进制	1101000	1100101	1101100	1101100	1101111

在 Go 语言中，可使用格式设置将十进制值转换为字符和二进制表示。

```
s := "hello"
fmt.Println("%q", s[0])
// outputs 'h'
fmt.Println("%b", s[0])
// outputs 1101000
```

很多国际字符都不止 1 字节，在下面的示例中，每个字符的长度都是 3 字节。

```
s := "こんにちは"
fmt.Println(len(s))
// outputs 15
```

即便不完全明白二进制、字节和字符字面量，也没有关系。请注意，在 Go 语言中，字符串实际上是字节切片，这意味着可以像操作其他字节切片一样操作字符串。

9.3.3 处理字符串

给字符串变量赋值后，就可使用标准库中的 strings 包提供的任何方法。这个包提供了一

套完备的字符串处理函数，其文档非常详尽。下面介绍几个字符串处理示例，但要全面了解strings包提供的函数，建议您阅读文档。

1. 将字符串转换为小写

函数ToLower接受一个字符串，并将其中所有的大写字符都转换为小写，如程序清单9.9所示。

程序清单9.9 将字符串转换为小写

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "strings"
 6:  )
 7:
 8:  func main() {
 9:      fmt.Println(strings.ToLower("VERY IMPORTANT MESSAGE"))
10:  }
```

TRY IT YOURSELF ▼

将字符串转换为小写

在这个示例中，您将明白如何将字符串转换为小写。

1. 在文本编辑器中打开文件hour09/example09.go，并尝试理解它是做什么的。
2. 在终端中使用命令go run example09.go运行这个程序。
3. 您将在控制台中看到小写的字符串。

```
very important message
```

▲

2. 在字符串中查找子串

处理字符串时，另一个常见的任务是在字符串中查找子串。方法Index提供了这样的功能，它接受的第二个参数是要查找的子串。如果找到，就返回第一个子串的索引号；如果没有找到，就返回-1。别忘了，索引从0开始！如程序清单9.10所示。

程序清单9.10 查找子串

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "strings"
 6:  )
 7:
 8:  func main() {
 9:      fmt.Println(strings.Index("surface", "face"))
10:      fmt.Println(strings.Index("moon", "aer"))
11:  }
```

TRY IT YOURSELF

在字符串中查找子串

在这个示例中，您将明白如何在一个字符串中查找另一个字符串。

1. 在文本编辑器中打开文件 hour09/example10.go，尝试理解它是做什么的。
2. 在终端中使用命令 go run example10.go 运行这个程序。
3. 您将看到第一个子串始于索引 3，而第二个子串没有找到。

```
3
-1
```

3. 删除字符串中的空格

strings 包提供了很多将字符串的某些部分删除的方法。处理来自用户或数据源的输入时，一种常见的任务是确保开头和末尾没有空格。方法 TrimSpace 提供了这样的功能，如程序清单 9.11 所示。

程序清单 9.11 删除开头和末尾的空白

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "strings"
 6:  )
 7:
 8:  func main() {
 9:      fmt.Println(strings.TrimSpace(" I don't need all this space ")
10:  }
```

TRY IT YOURSELF

删除空白

在这个示例中，您将明白如何删除字符串开头和末尾的空白。

1. 在文本编辑器中打开文件 hour09/example11.go，尝试理解它是做什么的。
2. 在终端中使用命令 go run example11.go 运行这个程序。
3. 您将在控制台中看到开头和末尾的空格都被删除后的字符串。

```
I don't need all this space
```

9.4 小结

本章介绍了 Go 语言中的字符串。您学习了解释型字符串字面量和原始字符串字面量之间的差别，熟悉了 rune 字面量以及如何使用它们来设置文本行的格式；您学习了如何使用运算符+和+=以及缓冲区来拼接字符串；您了解了字符串背后的原理，还有 UTF-8 及其在 Go 语言中的工作原理；最后，您学习了如何使用标准库中的 strings 包来操作字符串。

9.5 问与答

问：名称“字符串”是怎么来的？

答：虽然没有权威答案，但之所以称为“字符串”，是因为它是一系列（一连串）表示字符的字节。

问：既然 Go 语言支持 UTF-8，我能在代码中使用除英语外的其他语言吗？

答：可以。Go 语言代码被解读为 UTF-8，这意味着函数和方法的名称可包含 UTF-8 支持的任何字符。

问：创建字符串后，可对其进行修改吗？

答：在 Go 语言中，字符串是不可变的，这意味着创建后就不能修改。如果您试图重新声明变量，将导致编译错误。您可使用复合赋值运算符+=来拼接字符串。

9.6 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

9.6.1 小测验

1. 解释型字符串字面量和原始字符串字面量有何不同？
2. 表示制表符的 rune 字面量字符是什么？如何在解释型字符串字面量中使用它。
3. 通过执行大量拼接操作来创建字符串时，哪种方式的效率最高？
4. 为何推出 UTF-8？

9.6.2 答案

1. 解释型字符串字面量使用双引号（"），可包含 rune 字面量，这意味着可使用特殊字符来设置格式；原始字符串字面量使用反引号（`），位于反引号中的格式保持不变，这包括空格、制表符和换行符。

2．制表符用\t 表示。当用于（用双引号括起的）解释型字符串字面量中时，\t 将在字符串中添加一个制表符。

3．同构执行大量拼接操作来创建字符串时，效率最高的方式是使用字节缓冲区。Herman Schaaf 执行了一系列基准测试，大致确定了在 Go 语言中执行字符串拼接的各种方法的性能。

4．推出 UTF-8 旨在提供一种能够表示地球上几乎所有字符的标准方式。

9.7 练习

编写一个简短的程序，它接受字符串“Oh I do like to be beside the seaside”，并打印如下内容。

- 将这个字符串转换为大写后的结果。
- 将其中的“seaside”替换为“bar”后的结果。
- 单词 the 在这个字符串中的位置（索引号）。

Did you Know?

提示： 需要用到的所有方法都位于标准库的 strings 包中。

第 10 章

处理错误

本章介绍如下内容。

- 错误处理及 Go 语言的独特之处。
- 理解错误类型。
- 创建错误。
- 设置错误的格式。
- 从函数返回错误。
- 错误和可用性。
- 慎用 **panic**。

软件不可避免地会有错误及遇到未考虑到的情形，很多语言选择在发生必须捕获的错误时引发异常，而 Go 语言处理错误的方式很有趣——将错误作为一种类型，这意味着可将错误传递给函数和方法。本章介绍 Go 语言中的错误处理以及如何利用错误处理。

提示： 未雨绸缪！

优秀的程序员居安思危，他们会考虑可能出现的各种问题，进而编写好代码来处理灾难！很多人都在准备面试时考虑了所有可能出现问题的地方，这样就不会由于公共汽车老不来而误了面试，因为早就准备了 B 计划。

Did you Know?

要编写健壮、可靠而易于维护的代码，错误处理至关重要。如果您是为自己编写代码，考虑到意外的情形是十分必要的；如果您是编写供他人使用的库或包，错误处理对创建健壮、可靠而值得信任的代码至关重要。

10.1 错误处理及 Go 语言的独特之处

在 Go 语言中，一种约定是在调用可能出现问题的方法或函数时，返回一个类型为错误

的值。这意味着如果出现问题，函数通常不会引发异常，而让调用者决定如何处理错误。程序清单 10.1 演示了这种约定，其中使用的函数 ioutil.Readfile 在出现问题时返回一个错误值。

程序清单 10.1 读取文件时处理错误

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "io/ioutil"
 6:  )
 7:
 8:  func main() {
 9:      file, err := ioutil.ReadFile("foo.txt")
10:      if err != nil {
11:          fmt.Println(err)
12:          return
13:      }
14:      fmt.Println("%s", file)
15:  }
```

在没有文件 foo.txt 的系统中运行这个程序时，将触发错误。

```
go run example01.go
open foo.txt:  no such file or directory
```

这个示例包含的代码不多，却展示了 Go 语言对错误的众多看法。

- 使用标准库中 io/ioutil 包内的函数 ReadFile 读取文件。
- 如果有错误，就意味着返回的错误值不为 nil。
- 打印错误，程序就此结束。
- 如果没有错误，就打印文件的内容。

要理解 Go 语言处理错误的方式，最重要的是明白函数 ReadFile 接受一个字符串参数，并返回一个字节切片和一个错误值。这个函数的定义如下。

```
func ReadFile(filename string) ([]byte, error)
```

这意味着函数 ReadFile 总是会返回一个错误值，可对其进行检查。在前述示例的 main 函数中，将方法 ReadFile 的返回值存储到了两个变量（file 和 err）中，这是 Go 代码中常见的模式，您经常会见到。

```
file, err := ioutil.ReadFile("foo.txt")
```

语法:=是 Go 语言中的简短赋值语句，只能在函数中使用。Go 编译器会自动推断出变量的类型，而无须显式地声明，因此这里无须告诉编译器 file 是一个字节切片，而 err 是一个错误。这提供了极大的便利，前述代码与下面的代码等价。

```
var file []byte
var err error
file, err = ioutil.Readfile("foo.txt")
```

在 Go 语言中，有一种约定是，如果没有发生错误，返回的错误值将为 nil。这让程序员调用方法或函数时，能够检查它是否像预期那样执行完毕。

```
if err != nil {
    // something went wrong
}
```

在 Go 程序中，这种做法很常见。有些开发人员认为，这种做法很繁琐，因为它要求调用每个方法或函数时都检查错误，导致代码重复。

这说得也许没错，但 Go 语言处理错误的方式比其他语言更灵活，因为可像其他类型一样在函数之间传递错误。这通常意味着代码要简短得多。如果您要更深入地了解这一点，可阅读 Rob Pike 的博文《Errors are Values》。

TRY IT YOURSELF ▼

Go 语言中的错误处理方式

在这个示例中，您将运行一个返回错误的程序。

1. 在文本编辑器中打开文件 hour10/example01.go，尝试理解它是做什么的。如果需要，请参阅前面的解读。
2. 在终端中使用命令 go run example01.go 运行这个程序。
3. 您将看到如下消息。

```
open foo.txt:  no such file or directory
```

▲

10.2 理解错误类型

在 Go 语言中，错误是一个值。标准库声明了接口 error，如下所示。

```
type error interface {
    Error() string
}
```

这个接口只有一个方法——Error，它返回一个字符串。

10.3 创建错误

您已知道如何使用错误，但如果要创建并返回错误，该怎么办呢？标准库中的 errors 包支持创建和操作错误。

程序清单 10.2 演示了如何创建并打印错误。

程序清单 10.2 创建并打印错误

```
1:  package main
2:
3:  import (
4:      "fmt"
```

```
 5:  )
 6:
 7:  func main() {
 8:      err := errors.New("Something went wrong")
 9:      if err != nil {
10:          fmt.Println(err)
11:      }
12:  }
```

如果您运行这个示例，将打印如下错误。

```
go run example02.go
Something went wrong
```

这个示例演示了如何创建并检查错误，对其中的代码解读如下。

- 使用 errors 包中的方法 New 创建一个错误。
- 使用 if 语句检查这个错误值是否为 nil。
- 如果不为 nil，就将它打印出来。

TRY IT YOURSELF

创建错误

在这个示例中，您将明白如何创建错误。

1. 在文本编辑器中打开文件 hour10/example02.go，尝试理解它是做什么的。如果需要，请参阅前面对这些代码的解读。
2. 在终端中使用命令 go run example02.go 运行这个程序。
3. 您将看到如下消息。

```
Something went wrong
```

10.4 设置错误的格式

除 errors 包外，标准库中的 fmt 包还提供了方法 Errorf，可用于设置返回的错误字符串的格式，如程序清单 10.3 所示。这能够让您将多个值合并成更有意义的错误字符串，从而动态地创建错误字符串。

程序清单 10.3 使用 fmt 包设置错误字符串的格式

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      name, role := "Richard Jupp", "Drummer"
```

```
 9:      err := fmt.Errorf("The %v %v quit", role, name)
10:      if err != nil {
11:          fmt.Println(err)
12:      }
13:  }
```

如果您运行这个程序，将打印如下错误字符串。

```
go run example03.go
The Drummer Richard Jupp quit
```

TRY IT YOURSELF ▼

设置错误字符串的格式

在这个示例中，您将明白如何设置错误字符串的格式。

1. 在文本编辑器中打开文件 hour10/example03，尝试理解它是做什么的。
2. 在终端中使用命令 go run example03.go 运行这个程序。
3. 您将看到一条消息。

```
The Drummer Richard Jupp quit
```

4. 修改返回值并再次运行这个程序。您明白使用变量有助于让这条消息变成动态的了吗？

▲

10.5 从函数返回错误

本章开头说过，Go 语言的做法是从函数和方法中返回一个错误值。前面介绍了如何创建并返回错误，程序清单 10.4 是一个这样的示例。

程序清单 10.4 返回错误

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func Half(numberToHalf int) (int, error) {
 8:      if numberToHalf%2 != 0 {
 9:          return -1, fmt.Errorf("Cannot half %v", numberToHalf)
10:      }
11:      return numberToHalf / 2, nil
12:  }
13:
14:  func main() {
15:      n, err := Half(19)
16:      if err != nil {
17:          fmt.Println(err)
18:          return
```

```
19:         }
20:         fmt.Println(n)
21:     }
```

在这个示例中，一个函数返回错误，而调用者对其进行处理。对其中的代码解读如下。

- 将整数值 19 传递给函数 Half。
- 函数 Half 检查这个值是否为偶数。这里使用了 Go 语言中的求模运算符，它返回除法运算的余数。如果余数不为零，传入的值就不可能是偶数。在这里，余数确实不为零。
- 函数 Half 返回-1 和一个错误值。
- 调用者检查是否有错误，这里发现确实有错误。
- 打印这个错误，程序到此结束。

如果您运行这个程序，将打印一个错误。

```
go run function.go
Cannot half 19
```

这个示例演示了 Go 语言错误处理方式的一个优点：错误处理不是在函数中，而是在调用函数的地方进行的。这在错误处理方面提供了极大的灵活性，而不是简单地一刀切。

10.6 错误和可用性

除从技术角度考虑 Go 语言的错误处理方式和错误生成方式外，还需从以用户为中心的角度考虑错误。编写供他人使用的库或包时，您编写和使用错误的方式将极大地影响可用性。使用您编写的库时，用户可能遇到错误，并尝试从错误中恢复。请看下面的错误，您认为自己能够轻松地处理它或从中恢复吗？

```
You broke something! Good luck!!
```

这条错误消息很糟糕，因为它没有提供有关哪里出现问题的线索，也没有提供有关如何恢复的建议。下面的错误消息更好。

```
No config file found. Please create one at ~/.foorc.
```

为什么这样说呢？有如下几个原因。

- 具体地指出了问题所在。
- 提供了问题解决方案。
- 对用户更有礼貌。

如果库用户相信错误会以一致的方式返回，且包含有用的错误消息，则用户能够从错误中恢复的可能性将高得多。他们很可能也会认为您编写的库不仅很有用，而且值得信任。

10.7 慎用 panic

panic 是 Go 语言中的一个内置函数，它终止正常的控制流程并引发恐慌（panicking），导致程序停止执行。出现普通错误时，并不提倡这种做法，因为程序将停止执行，并且没有

任何回旋余地。程序清单 10.5 表明，panic 导致程序立即停止执行。

程序清单 10.5 使用 panic 来停止执行程序

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      fmt.Println("This is executed")
 9:      panic("Oh no. I can do no more. Goodbye.")
10:      fmt.Println("This is not executed")
11:  }
```

运行这个示例将引发 panic，导致程序崩溃。

```
go run panic.go
This is executed
panic:  Oh no. I can do no more. Goodbye.

goroutine 1 [running]:
panic(0x48a560, 0xc42000a320)
    /usr/lib/go/src/runtime/panic.go:500 +0x1a1
    main.main()
    /home/go/golang-book-examples/hour10 /example05.go:7 +0xef
    exit status 2
```

调用 panic 后，程序将停止执行，因此打印 This is not executed 的代码行根本没有机会执行。

在 Go 代码中，常常滥用下面的做法，这实际是说：朋友，咱们无路可走，只能让程序崩溃了。在有些情况下，这样做是合适的，但通常不应这样做。

```
if err != nil {
    panic(err)
}
```

在下面的情形下，使用 panic 可能是正确的选择。

- 程序处于无法恢复的状态。这可能意味着无路可走了，或者再往下执行程序将带来更多的问题。在这种情况下，最佳的选择是让程序崩溃。
- 发生了无法处理的错误。

TRY IT YOURSELF ▼

使用 panic

在这个示例中，您将明白如何使用 panic。

1. 在文本编辑器中打开文件 hour10/example05.go，尝试理解它是做什么的。
2. 在终端中使用命令 go run example05.go 运行这个程序。
3. 注意到出现了 panics，导致程序停止执行。

▲

10.8 小结

本章介绍了 Go 语言处理错误的方式。您学习了如何使用错误，再接着学习了如何创建错误；接下来，您了解了 Go 语言独特的错误处理方式——由调用者负责处理错误；然后，您获悉除从技术角度了解错误的工作原理外，还应考虑错误使用方式并让调用者能够从错误中恢复；最后，您了解到 panic 会导致程序立即停止执行，因此应尽量少用。

10.9 问与答

问：在 Go 代码中，if err != nil 随处可见。这看起来重复太多，请问这种做法是最佳的吗？

答：虽然这看起来重复太多，但能够检查错误实际上是 Go 语言的特色，这给程序员提供了极大的控制权。虽然可采用办法和第三方包来减少这样的重复，但大多数 Go 程序员都喜欢能够随心所欲地处理错误。

问：真的应该关心错误吗？

答：绝对应该！要编写富有弹性而稳定的代码，处理错误至关重要。对程序可能出现的错误以及如何从中恢复这些问题考虑得越多，代码的质量就越好。

问：Go 支持异常吗？

答：不同于 Java 等其他语言，Go 语言不支持传统的 try-catch-finally 控制结构，而向函数或方法的调用者报告错误。Go 语言将错误作为返回值，这只是它做出的一种设计决策，但对这种处理错误的方式仍存在争议。

10.10 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

10.10.1 小测验

1. Go 语言处理错误的方式有何独特之处？
2. 在 Go 语言中，可在错误中使用变量吗？
3. 在什么情况下该使用 panic？

10.10.2 答案

1. 在 Go 语言中，可从函数返回多个值，一种约定是将错误作为最后一个返回值。这样可自定义错误并将其返回给调用者，由调用者决定如何处理错误。其他语言认为，不管发生什么错误，都应让程序立即崩溃；而 Go 语言更灵活，它让方法或函数的调用者决定如何处理错误。

2．是的。事实上，可使用任何代码来生成错误，再返回它。在博文《Errors are Values》中，Rob Pike 阐述了如下观点。

“可对值进行编程，而错误是值，因此可对错误进行编程。错误不像异常，因为错误没什么特别之处，而未处理的异常可能导致程序崩溃。”

3．仅当程序无法从错误中恢复时，才使用 panic。panic 是没有办法的办法，仅当让程序崩溃是最负责任的选择时才应使用 panic。

10.11 练习

1．在您编写程序或使用计算机的过程中，是否遇到过错误让您沮丧的情形？这样的沮丧表现在什么方面呢？

2．在网上搜索用于读取并分析 toml 格式文件的第三方包。通过阅读其主页，您知道这个包的功能和用法吗？请安装这个包，并创建一个简单的程序以证明您明白了其功能和用法。

第 11 章

使用 Goroutine

本章介绍如下内容。

- 理解并发。
- 并发和并行。
- 通过 Web 浏览器来理解并发。
- 阻塞和非阻塞代码。
- 使用 Goroutine 处理并发操作。
- 定义 Goroutine。

本章介绍并发和 Goroutine。您将明白顺序执行和并发执行的差别，知道 Goroutine 是应对网络延迟的方式之一；您将了解并发和并行的差别，知道 Goroutine 是如何提高程序的速度的。Goroutine 是最优雅的 Go 语言功能之一，如果您以为这很复杂，那么请接着往下阅读。Go 使用一个关键字就解决了这个问题！

11.1 理解并发

要理解 Goroutine，必须先明白并发的含义。在最简单的计算机程序中，操作是依次执行的，执行顺序与出现顺序相同。想一想脚本中的代码行吧，这些代码行按出现顺序执行，一行执行完毕后才执行下一行。对很多程序来说，这种行为是可取的，程序员只要知道执行完一行代码后才会执行下一行，就能准确地推断出脚本的逻辑。

这种行为很像餐厅服务员给顾客点菜。服务员必须先将菜单给顾客，然后顾客看过菜单并点菜完毕后，服务员才能将点菜单交给厨师。服务员给顾客提供服务的过程大致如下。

1. 将菜单递给顾客。
2. 接收顾客的点菜单。
3. 将点菜单交给厨师。

4. 从厨师那里取菜。

5. 将菜交给顾客。

为这个过程编写程序时，完全可以认为一个任务完成后才能接着执行下一个任务。这个过程很像按出现顺序执行脚本中的代码——一行代码执行完毕后再执行下一行。

对很多编程任务而言，按顺序执行任务的理念不仅可行，而且效果很显著。下面是一些这样的例子。

- 基于轮次的简单终端游戏。
- 温度转换器。
- 随机数生成器。

另一种理念是不必等到一个操作执行完毕后再执行下一个，编程任务和编程环境越复杂，这种理念就越重要。提出这种理念旨在让程序能够应对更复杂的情形，避免执行完一行代码后再执行下一行，从而提高程序的执行速度。程序完全按顺序执行时，如果某行代码需要很长时间才能执行完毕，那么整个程序将可能因此而停止，导致用户长时间等待事件的发生。

现代编程必须考虑众多时间不可预测的变数。例如，您无法确定网络调用需要多长时间才能完成，也无法确定读取磁盘文件需要多长时间。

假设程序需要从天气服务那里获取某个地方当前的天气情况时，就需要编写一些代码来执行这种请求并处理 Web 服务器的响应。程序发出请求后，很多因素都可能影响响应返回的速度，如以下几种。

- 查找天气服务地址的 DNS 的速度。
- 程序和天气服务器之间网络连接的速度。
- 建立与天气服务器连接的速度。
- 天气服务器的响应速度。

鉴于所有这些因素都不是发出请求的程序能够控制的，因此完全有理由认为响应速度是无法预测的。另外，每次请求得到响应的时间都可能不同。面对这样的情形，程序员可选择等待响应——阻塞程序直到响应返回为止，也可继续执行其他有用的任务。大多数现代编程语言都提供了选择空间，让程序员可等待响应，也可继续做其他事情。

回过头来看餐厅服务员给顾客提供服务的过程，完成其中每个步骤的时间都是不确定的。

- 顾客需要多长时间才能入座？
- 顾客需要多长时间才能点好菜？
- 顾客需要多长时间才能下单？
- 厨师需要多长时间才开始做菜？
- 厨师需要多长时间才能将菜做好？

如果按顺序做，服务员能够很好地为顾客服务，但无法同时为其他顾客提供服务！如果每位服务员都专为一位顾客服务，则餐厅的收费将非常高。相反，服务员可并发地执行任务。这意味着在厨师做菜时服务员可以给其他顾客点菜，并在其他顾客点菜期间去取菜。

在现实世界中，很多事情都是同时进行的：乘客在上车的同时听音乐，在排队时读书。鉴于此，编程语言有必要提供模拟这种情形的方式。

随着互联网的日益普及，网络编程越来越常见：程序可能向多个服务请求信息；数据库可能位于另一个完全不同的网络中。鉴于一切都是基于网络的，要可靠地预测任务完成的时间是很难做到的。

11.2　并发和并行

理解并发后，该讨论 Goroutine 了，但在此之前先来说说并发和并行的差别，这一点很重要。我们将以为生日聚会烘培 100 个蛋挞为例，来说明并发和并行的差别。咱们假设蛋糕粉和烘烤托盘多得不得了，而我们的目标是尽快将蛋挞烤好。

如果采用顺序方式，就意味着每次只能在一个烤箱中烤一个蛋挞。这种做法的效率显然很低，因为这将需要很长时间：烤好一个蛋挞后再将另一个蛋挞放入烤箱。另外，时间上也无法预测，因为有的蛋挞烤得快，有的烤得慢；有的需要的时间短点，有的需要的时间长点。

一种并发方式是，使用烘烤托盘每次烤多个蛋挞。这样做的效率要高得多，但还是不能同时烤好所有的蛋挞。例如，根据蛋挞的大小和所处烤箱的位置，烤好每个蛋挞的时间可能不同。相比于顺序方法，这种做法的速度更快，快多少取决于可同时烘烤多少个蛋挞。

并发方法的速度受制于众多因素，其中一个是烤箱的尺寸。如果有位朋友家也有烤箱，就可两家同时烤，从而进一步提高效率。同时烤多个蛋挞被称为并发；而将烤蛋挞的任务分为两部分，由两家分别烤，烤好后再放在一起，这被称为并行。以并行的方式执行任务时，可利用并发性，也可不利用；它相当于将工作分成多个部分，各部分的工作完成后再将结果合并。

这好像很复杂（涉及复杂的计算机科学），但 Go 语言的设计者之一 Rob Pike 很好地总结了并发和并行的差别：

并发就是同时处理很多事情，而并行就是同时做很多事情。

在现代编程中，并发是不可或缺的部分。对有些程序来说，并发至关重要（在确保性能方面尤其如此）。下面是一些这样的程序。

- 聊天程序。
- 多玩家游戏。
- Web 服务器。
- 从磁盘读取数据。

Google 日常工作的并发需求是催生出 Go 的动力之一，因为使用传统的系统语言难以编写高效的并发代码。

11.3　通过 Web 浏览器来理解并发

大家每天都要使用 Web 浏览器，而网站的加载速度通常很快，这都是拜并发性所赐。为合成网页并将其显示给用户，浏览器背后的技术必须做大量的并发工作。通常，网页由图像

和脚本组成，而这些图像和脚本来自网上众多不同的服务器。

为理解并发性，您将进入浏览器的幕后，观看网页的合成过程。为此，可使用浏览器的开发者工具。在 Google Chrome 中，开发者工具可在菜单 Developer>Network 中找到；如果您使用的是其他浏览器，但不知道如何打开开发者工具，可在 Google 中输入“开发者工具”和浏览器的名称。

打开开发者工具后，在地址栏中输入 BBC 主页网址并按回车。在页面加载期间，观看发送的请求。您将能够看到所有的请求。对于每个请求，您可知道它花了多长时间以及这些时间都花在什么地方。就本章的目的而言，开发者提供的信息太过详细了，但这里的要点在于，Web 浏览器不依次发出请求，而是同时发出请求，以尽快渲染页面或其组成部分。这样做的结果是，页面的加载速度在用户看来很快。仅当所有请求都结束后，页面才加载完毕，但在此之前浏览器依然能够做很多有用的事情。例如，图像加载后，浏览器就可将其渲染到页面上，如图 11.1 所示。

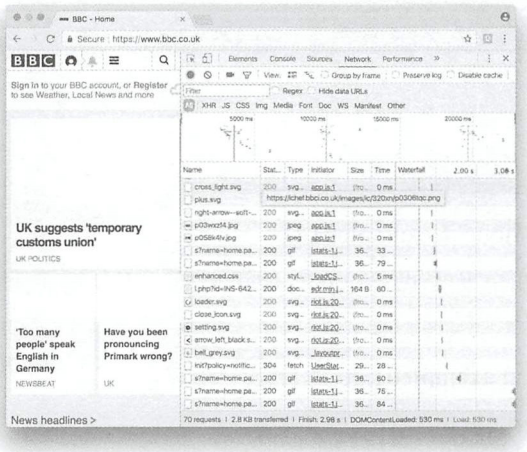


图 11.1
加载 BBC 主页

11.4 阻塞和非阻塞代码

基于对并发性的大致认识，下面来编写一个程序，模拟函数调用阻塞程序的执行直到操作完成的情形。为模拟缓慢的函数调用，可使用 time.Sleep，它的作用是让程序暂停指定的时间。在实际编程中，这可能是缓慢的函数调用或需要运行很长时间的函数。

程序清单 11.1 是一个模拟阻塞的函数调用的程序。

程序清单 11.1 模拟阻塞的函数调用

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func slowFunc() {
 9:      time.Sleep(time.Second * 2)
10:      fmt.Println("sleeper() finished")
11:  }
12:
13:  func main() {
```

```
14:       slowFunc()
15:       fmt.Println("I am not shown until slowFunc() completes")
16:   }
```

对程序清单 11.1 解读如下。

- 这个程序执行时，将调用执行方法 time.Sleep 的函数 slowFunc。
- 方法 time.Sleep 让程序暂停 2s。
- 程序暂停时不会执行其他代码，因此暂停期间不会执行函数 main 中的第二行代码。
- 2s 后，函数 slowFunc 打印一行文本再返回。
- 控制权返回到 main 函数，因此执行第二行代码——向终端打印一条消息。

这些代码是阻塞的，因为等待函数 slowFunc()返回期间，没有执行其他代码。

▼TRY IT YOURSELF

阻塞的代码

在这个示例中，您将明白阻塞的代码。

1. 打开本书代码示例中的 hour11/example01.go。
2. 阅读这些代码，尝试理解它们是做什么的。如果需要，请参阅前面对程序清单 11.1 的解读。
3. 在终端中使用命令 go run example01.go 运行这个程序。
4. 暂停一段时间后，您将在终端中看到如下文本。

```
sleeper() finished
I am not shown until slowFunc() completes
```

▲

11.5 使用 Goroutine 处理并发操作

Go 语言提供了 Goroutine，让您能够处理并发操作。在程序清单 11.1 中，通过使用 Goroutine，可在调用函数 slowFunc 后立即执行 main 函数中的第二行代码。在这种情况下，函数 slowFunc 依然会执行，但不会阻塞程序中其他代码行的执行。

Goroutine 使用起来非常简单，只需在要让 Goroutine 执行的函数或方法前加上关键字 go 即可。可对程序清单 11.1 进行修改，以使用 Goroutine，如程序清单 11.2 所示。

程序清单 11.2 使用 Goroutine

```
1:  package main
2:
3:  import (
4:      "fmt"
5:      "time"
6:  )
7:
```

```
 8:  func slowFunc() {
 9:      time.Sleep(time.Second * 2)
10:      fmt.Println("sleeper() finished")
11:  }
12:
13:  func main() {
14:      go slowFunc()
15:      fmt.Println("I am now shown straightaway!")
16:  }
```

然而，执行这些代码时，结果令人意外。

TRY IT YOURSELF ▼

理解 Goroutine 在何时返回

在这个示例中，您将明白 Goroutine 在何时返回。

1. 打开本书代码示例中的 hour11/example02.go。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example02.go 运行这个程序。
4. 您将马上在终端中看到如下文本。

```
I am now shown straightaway!
```

▲

你根本看不到调用 slowFunc 的结果，这是因为 Goroutine 立即返回，这意味着程序将接着执行后面的代码，然后退出。如果没有其他因素阻止，则程序将在 Goroutine 返回前就退出。第 12 章将介绍如何使用通道来管理 Goroutine，但这里将再添加一个 time.Sleep 调用来阻止程序立即退出。程序清单 11.3 演示了如何使用 Goroutine 来实现并发执行。

程序清单 11.3　演示如何使用 Goroutine 来实现并发执行

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func slowFunc() {
 9:      time.Sleep(time.Second * 2)
10:      fmt.Println("sleeper() finished")
11:  }
12:
13:  func main() {
14:      go slowFunc()
15:      fmt.Println("I am not shown until slowFunc() completes")
16:      time.Sleep(time.Second * 3)
17:  }
```

如果您运行程序清单 11.3，将发现其行为符合预期：在 slowFunc 返回前继续执行程序中的其他代码。

▼ TRY IT YOURSELF

理解如何使用 Goroutine 来编写非阻断代码

在这个示例中，您将明白如何使用 Goroutine 来编写非阻断代码。

1. 打开本书代码示例中的 hour11/example03.go。
2. 阅读这些代码，尝试理解白它们是做什么的。
3. 在终端中使用命令 go run example03.go 运行这个程序。
4. 您将马上在终端中看到如下文本。

```
I am now shown straightaway!
```

5. 过段时间后，您将在终端中看到如下文本。

```
slowFunc() finished
```

▲

11.6 定义 Goroutine

如果您使用过其他编程语言，就会知道并发是编程语言提供的一种常见功能。例如，Node.js 使用事件循环来管理并发，而 Java 使用线程。Apache 和 Nginx 等 Web 服务器也使用不同的并发方法，Apache 喜欢使用线程和进程，而 Nginx 使用事件循环。如果您不明白这些术语，也不用担心。这里的重点是，实现并发的方式有很多，它们以不同的方式使用计算机资源，这使得编写可靠的软件或难或易。

与 Java 一样，Go 在幕后使用线程来管理并发，但 Goroutine 让程序员无须直接管理线程，它消除了这样做的痛苦。创建一个 Goroutine 只需占用几 KB 的内存，因此即便创建数千个 Goroutine 也不会耗尽内存。另外，创建和销毁 Goroutine 的效率也非常高。

Goroutine 是一个并发抽象，因此开发人员通常无须准确地知道操作系统中发生的情况。

11.7 小结

本章介绍了并发以及最优雅的 Go 语言功能之一——Goroutine。您了解到，现实世界中很多事情都是同时发生的，而现代编程在处理并发方面面临着众多挑战；您知道使用 Goroutine 来编写并发代码易如反掌，并学习了如何使用 Goroutine 来管理网络延迟。

11.8 问与答

问：Goroutine 看起来很神奇！我只需知道如何使用关键字 go 吗？

答：要使用 Goroutine，只需一个关键字就可搞定。比较难理解的是并发是什么、它对编

程有何影响。您可更深入地了解 Goroutine 的技术实现，但这在很大程度上说是 Go 语言设计者考虑的问题。

问：Goroutine 为何要立即返回？

答：Goroutine 之所以立即返回，是因为这样才符合非阻塞执行的理念。第 12 行将介绍通道——一种联系和管理并发 Goroutine 的方式。

问：可在哪些情况下使用 Goroutine？

答：在事件发生顺序未知的情况下，使用 Goroutine 是不错的选择。比如网络调用、读取磁盘文件以及创建事件驱动的程序（如聊天应用和游戏）。

11.9 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

11.9.1 小测验

1. 相比于依次执行，并发执行有哪些优点？
2. 何为阻塞式代码？
3. 除管理网络延迟外，还有哪些情况下并发编程很有用？

11.9.2 答案

1. 并发地执行代码意味着程序可能能够更快地执行完毕，并在数据就绪后就返回它，而无须等待程序其他部分结束。

2. 阻塞式代码让程序暂停执行。在本章的示例中，使用了 Time.Sleep 来暂停执行程序。如果不使用并发编程，速度缓慢的函数和性能不佳的代码则可能阻塞程序执行，导致程序缓慢。为避免阻塞式代码，方法之一是使用 Goroutine，因为它在结果就绪后将其返回，而不会阻塞程序的执行。

3. 使用并发编程的其他情形包括从磁盘文件读写数据、从网络读写数据以及从数据库读写数据。

11.10 练习

如果您有时间，请观看 Go 语言设计者之一 Rob Pike 的演讲视频“Concurrency Is Not Parallelism”。这个视频长约 30min，它很好地介绍了并发以及 Go 语言实现并发的方式。

第 12 章

通道简介

本章介绍如下内容。

- 使用通道。
- 使用缓冲通道。
- 阻塞和流程控制。
- 将通道用作函数参数。
- 使用 **select** 语句。
- 退出通道。

第 11 章介绍了 Goroutine，这是一种处理并行操作的方式。您知道，可通过编程并行地执行任务，并在响应返回时立即对其进行处理。本章介绍通道，它让您能够管理 Goroutine 之间的通信。通道和 Goroutine 一道提供了一个受控的环境，能够让您开发并发软件。

12.1 使用通道

如果说 Goroutine 是一种支持并发编程的方式，那么通道就是一种与 Goroutine 通信的方式。通道让数据能够进入和离开 Goroutine，可方便 Goroutine 之间进行通信。《Effective Go》有一句话很好地说明了 Go 语言的并发实现理念：

“不要通过共享内存来通信，而通过通信来共享内存。”

这句话说明了 Go 语言并发实现方式的不同之处，这部分有必要做进一步解释。在其他编程语言中，并发编程通常是通过在多个进程或线程之间共享内存实现的。共享内存能够让程序同步，确保程序以合乎逻辑的方式执行。在程序执行过程中，进程或线程可能对共享内存加锁，以禁止其他进程或线程修改它。这合乎情理，因为如果在操作期间共享内存被其他进程修改，可能会带来灾难性后果——引发 Bug 或导致程序崩溃。通过这种方式给内存加锁，可确保它是独占的——只有一个进程或线程能够访问它。

这听起来可能太过抽象，我们来看一个例子：两个人持有一个联名账户，他们要同时从这个账户支付费用，但这两笔交易的总额超过了账户余额。如果两个交易同时进行且不加锁，则余额检查可能表明资金充足，但实际上资金不够；然而，如果第一个交易将账户加锁，则直到交易完成，都可避免这样的情况发生。对于简单的并发而言，这种方式看似合理，但如果联名账户有 20 个持有人，且他们经常使用这个账户进行交易呢？在这种情况下，加锁管理工作可能非常复杂。

共享内存和锁的管理工作并非那么容易，很多编程语言要求程序员对内存和内存管理有深入认识。即便是久经沙场的程序员也会遇到这样的情形，即为找出进程或线程争用共享内存而引发的 Bug，需要花费数天时间。在使用共享内存的并发环境中，如果不能始终知道程序的哪部分将先更新数据，那么将难以推断其中发生的情况。

虽然使用共享内存有其用武之地，但 Go 语言使用通道在 Goroutine 之间收发消息，避免了使用共享内存。严格地说，Goroutine 并不是线程，但您可将其视为线程，因为它们能够以非阻塞方式执行代码。在前面关于两人持有一个联合账户的例子中，如果使用 Goroutine，将在账户持有人之间打开一个通信通道，让他们能够通信并采取相应的措施。例如，一个交易可能向通道发送一条消息，而通道可能限制后续交易或另一个账户持有人的行为。通过收发消息，使得能够以推送方式协调并发事件。事件发生时，可将触发的消息推送给接收者。使用共享内存时，程序必须检查共享内存。在变化频繁的并发编程环境中，很多人都认为使用消息是一种更佳的通信方式。

第 11 章介绍了一个示例，其中使用了 Goroutine 来执行运行速度缓慢的函数，以免阻塞整个程序的执行，如程序清单 12.1 所示。

程序清单 12.1　简单的 Goroutine 示例

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func slowFunc() {
 9:      time.Sleep(time.Second * 2)
10:      fmt.Println("sleeper() finished")
11:  }
12:
13:  func main() {
14:      go slowFunc()
15:      fmt.Println("I am not shown until slowFunc() completes")
16:      time.Sleep(time.Second * 3)
17:
```

在程序清单 12.1 的第 9 行中，使用了一个定时器，旨在避免程序在 Goroutine 返回前退出。在介绍 Goroutine 的示例中，这样做完全可行，但在更复杂的并发程序中，使用定时器绝非好主意。为了管理 Goroutine 和并发，Go 语言提供了通道。在程序清单 12.1 中，如果能够在 Goroutine 和程序之间通信，并让 Goroutine 结束时能够告诉主程序就好了。这正是通道的用武之地。

通道的创建语法如下。

```
c := make(chan string)
```

对这种语法解读如下。

- 使用简短变量赋值，将变量 c 初始化为:=右边的值。
- 使用内置函数 make 创建一个通道，这是使用关键字 chan 指定的。
- 关键字 chan 后面的 string 指出这个通道将用于存储字符串数据，这意味着这个通道只能用于收发字符串值。

向通道发送消息的语法如下。

```
c <- "Hello World"
```

请注意其中的<-，这表示将右边的字符串发送给左边的通道。如果通道被指定为收发字符串，则只能向它发送字符串消息，如果向它发送其他类型的消息将导致错误。

从通道那里接收消息的语法如下。

```
msg := <-c
```

Did you Know?

提示：要从通道那里接收消息，需要在<-后面加上通道名。可使用简短变量赋值，将来自通道的消息直接赋给变量。箭头向左表示数据离开通道（接收），箭头向右表示数据进入通道（发送）。

从很大程度上说，了解通道创建语法以及消息收发语法后，就大致掌握了通道的用法。现在可对程序清单 12.1 进行修改以使用通道，如程序清单 12.2 所示。

程序清单 12.2 使用通道进行通信

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func slowFunc() {
 9:      time.Sleep(time.Second * 2)
10:      c <- "slowFunc() finished"
11:  }
12:
13:  func main() {
14:      c := make(chan string)
15:      go slowFunc()
16:
17:      msg := <-c
18:      fmt.Println(msg)
19:  }
```

对程序清单 12.2 解读如下。

- 创建一个存储字符串数据的通道，并将其赋给变量 c。
- 与程序清单 12.1 一样，使用了一个 Goroutine 来执行函数 slowFunc。
- 函数 slowFunc 将通道当作参数。
- slowFunc 函数的单个参数指定了一个通道和一个字符串的数据类型。
- 声明变量 msg，用于接收来自通道 c 的消息。这将阻塞进程直到收到消息为止，从

而避免进程过早退出。

- 函数 slowFunc 执行完毕后向通道 c 发送一条消息。
- 接收并打印这条消息。
- 由于没有其他的语句，因此程序就此退出。

TRY IT YOURSELF ▼

使用通道与 Goroutine 通信

在这个示例中，您将明白如何使用通道与 Goroutine 通信。

1. 打开本书代码示例中的 hour12/example01.go。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example01.go 运行这个程序。
4. 2s 后您将在终端中看到一条消息。

```
slowFunc() finished
```

▲

12.2 使用缓冲通道

通常，通道收到消息后就可将其发送给接收者，但有时候可能没有接收者。在这种情况下，可使用缓冲通道。缓冲意味着可将数据存储在通道中，等接收者准备就绪再交给它。要创建缓冲通道，可向内置函数 make 传递另一个表示缓冲区长度的参数。

```
messages := make(chan string, 2)
```

这些代码创建一个可存储两条消息的缓冲通道。现在可在这个通道中添加两条消息了——虽然没有接收者。请注意，缓冲通道最多只能存储指定数量的消息，如果向它发送更多的消息将导致错误。

```
messages <- "hello"
messages <- "world"
```

消息将存储在通道中，直到接收者准备就绪。程序清单 12.3 的示例中，在一个缓冲通道中添加了两条消息，并在接收者准备就绪后接收了这些消息。

程序清单 12.3 让缓冲通道接收两条消息

```
1:  package main
2:
3:  import (
4:      "fmt"
5:      "time"
6:  )
7:
8:  func slowFunc(c chan string) {
```

```
 9:      for msg := range c {
10:          fmt.Println(msg)
11:      }
12:  }
13:
14:  func main() {
15:      messages := make(chan string, 2)
16:      messages <- "hello"
17:      messages <- "world"
18:      close(messages)
19:      fmt.Println("Pushed two messages onto Channel with no receivers")
20:      time.Sleep(time.Second * 1)
21:      receiver(messages)
22:  }
```

在这个示例中，close 在以前没介绍过。它用来关闭通道，禁止再向通道发送消息。

```
close(messages)
```

对程序清单 12.3 解读如下。

- 创建一个长度为 2 的缓冲通道。
- 向通道发送两条消息。此时没有可用的接收者，因此消息被缓冲。
- 关闭通道，这意味着不能再向它发送消息。
- 程序打印一条消息，指出通道包含两条消息，再休眠 1s。
- 将通道作为参数传递给函数 receiver。
- 函数 receiver 使用 range 迭代通道，并将通道中缓冲的消息打印到控制台。

在知道需要启动多少个 Goroutine 或需要限制调度的工作量时，缓冲通道很有效。

▼ TRY IT YOURSELF

理解缓冲通道

在这个示例中，您将明白如何使用缓冲通道。

1. 打开本书代码示例中的 hour12/example02.go。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example02.go 运行这个程序。
4. 您将在终端中看到如下内容。

```
Pushed two messages onto Channel with no receivers
hello
world
```

▲

12.3 阻塞和流程控制

第 11 章说过，Goroutine 是 Go 语言提供的一种并发编程方式。速度缓慢的网络调用或函数

会阻塞程序的执行，而 Goroutine 能够让您对此进行管理。在并发编程中，通常应避免阻塞式操作，但有时需要让代码处于阻塞状态。例如，需要在后台运行的程序必须阻塞，这样才不会退出。

Goroutine 会立即返回（非阻塞），因此要让进程处于阻塞状态，必须采用一些流程控制技巧。例如，从通道接收并打印消息的程序需要阻塞，以免终止。

给通道指定消息接收者是一个阻塞操作，因为它将阻止函数返回，直到收到一条消息为止。程序清单 12.4 演示了阻塞和通道的一些微妙之处。您认为这个程序的输出是什么呢？

程序清单 12.4　通道和流程控制

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func slowFunc(c chan string) {
 9:      t := time.NewTicker(1 * time.Second)
10:      for {
11:          c <- "ping"
12:          <-t.C
13:      }
14:  }
15:
16:  func main() {
17:      messages := make(chan string)
18:      go pinger(messages)
19:      msg := <-messages
20:      fmt.Println(msg)
21:  }
```

TRY IT YOURSELF ▼

理解通道和流程控制

在这个示例中，您将明白通道和流程控制的一些微妙之处。

1. 打开本书代码示例中的 hour12/example03.go。
2. 阅读这些代码，尝试理解它们是做什么的。您认为这个程序的输出是什么呢？
3. 在终端中使用命令 go run example03.go 运行这个程序。
4. 您将在终端中看到如下内容。

```
ping
```

▲

如果您认为这个程序将在打印消息 ping 后退出，那么您说对了。收到一条消息后，阻塞操作将返回，而程序将退出。那么，如果创建不断监听通道中消息的监听器呢？这与通道的关系不大，而主要是与 Go 运行环境和执行流程相关。第 5 章介绍了 Go 语言控制流程，您学习了 for 语句。通过使用 for 语句，可永久性地阻塞进程，也可让阻塞时间持续特定的迭代次数。

通过在程序清单 12.4 中添加一条 for 语句，可不断从通道那里接收消息并将其打印到控制台，如程序清单 12.5 所示。

程序清单 12.5 不断地执行指定的操作

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func slowFunc(c chan string) {
 9:      t := time.NewTicker(1 * time.Second)
10:      for {
11:          c <- "ping"
12:          <-t.C
13:      }
14:  }
15:
16:  func main() {
17:      messages := make(chan string)
18:      go pinger(messages)
19:      for {
20:          msg := <-messages
21:          fmt.Println(msg)
22:      }
23:  }
```

TRY IT YOURSELF

放置进程退出

在这个示例中，您将明白如何放置进程退出。

1. 打开本书代码示例中的 hour12/example04.go。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example04.go 运行这个程序。
4. 您将在终端中不断看到如下输出。

```
ping
ping
ping
```

▲

如果要在接收指定数量的消息后结束，则可使用包含迭代器的 for 语句。指定的迭代次数完成后，进程将退出。

```
for i := 0; i < 5; i++ {
    msg := <-messages
    fmt.Println(msg)
}
```

Goroutine 是非阻塞的，因此如果程序要阻塞，以接收大量的消息或不断地重复某个过程，

必须使用其他流程控制技术。

12.4 将通道用作函数参数

您已在前面的示例中了解过，可将通道作为参数传递给函数，并在函数中向通道发送消息。要进一步指定在函数中如何使用传入的通道，可在传递通道时将其指定为只读、只写或读写的。指定通道是只读、只写、读写的语法差别不大。

```
func channelReader(messages <-chan string) {
    msg := <-messages
    fmt.Println(msg)
}

func channelWriter(messages chan<- string) {
    messages <- "Hello world"
}

func channelReaderAndWriter(messages chan string) {
     msg := <-messages
     fmt.Println(msg)
     messages <- "Hello world"
}
```

<-位于关键字 chan 左边时，表示通道在函数内是只读的；<-位于关键字 chan 右边时，表示通道在函数内是只写的；没有指定<-时，表示通道是可读写的。

通过指定通道访问权限，有助于确保通道中数据的完整性，还可指定程序的哪部分可向通道发送数据或接收来自通道的数据。

12.5 使用 select 语句

假设有多个 Goroutine，而程序将根据最先返回的 Goroutine 执行相应的操作，此时可使用 select 语句。select 语句类似于第 5 章介绍的 switch 语句，它为通道创建一系列接收者，并执行最先收到消息的接收者。

select 语句看起来和 switch 语句很像。

```
channel1 := make(chan string)
channel2 := make(chan string)

select {
    case msg1 := <-channel1:
        fmt.Println("received", msg1)
    case msg2 := <-channel2:
        fmt.Println("received", msg2)
}
```

如果从通道 channel1 那里收到了消息，将执行第一条 case 语句；如果从通道 channel2 那里收到了消息，将执行第二条 case 语句。具体执行哪条 case 语句，取决于消息到达的时间，哪条消息最先到达决定了将执行哪条 case 语句。通常，接下来收到的其他消息将被丢弃。收到一条消息后，select 语句将不再阻塞。

程序清单 12.6 演示了 select 语句。

程序清单 12.6 select 语句

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func ping1(c chan string) {
 9:      time.Sleep(time.Second * 1)
10:      c <- "ping on channel1"
11:  }
12:
13:  func ping2(c chan string) {
14:      time.Sleep(time.Second * 2)
15:      c <- "ping on channel2"
16:  }
17:
18:  func main() {
19:      channel1 := make(chan string)
20:      channel2 := make(chan string)
21:
22:      go ping1(channel1)
23:      go ping2(channel2)
24:
25:      select {
26:      case msg1 := <-channel1:
27:          fmt.Println("received", msg1)
28:      case msg2 := <-channel2:
29:          fmt.Println("received", msg2)
30:      }
31:  }
```

对程序清单 12.6 解读如下。

- 创建两个用于存储字符串数据的通道。
- 创建两个向这些通道发送消息的函数。为模拟函数的执行速度，第一个函数休眠了 1s，而第二个休眠了 2s。
- 启动两个 Goroutine，分别用于执行这些函数。
- select 语句创建了两个接收者，分别用于接收来自通道 channel1 和 channel2 的消息。
- 1s 后，函数 ping1 返回，并向通道 channel1 发送一条消息。
- 收到来自通道 channel1 的消息后，执行第一条 case 语句——向终端打印一条消息。
- 整个 select 语句就此结束，不再阻塞进程，因此程序退出。

▼TRY IT YOURSELF

结合使用 select 语句和通道

在这个示例中，您将明白如何使用 select 语句。

1. 打开本书代码示例中的 hour12/example05.go。

2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example05.go 运行这个程序。
4. 您将在终端中看到如下内容。

```
received ping on channel1
```

可修改程序清单 12.5，以证明 select 语句将执行第一个满足条件的 case 语句。为此，可让函数 ping2 先返回——将函数 ping1 的休眠时间改为 3s。这样，将先向通道 channel2 发送消息，因此将执行第二条 case 语句（msg2）。

TRY IT YOURSELF

理解 select 如何选择要执行的 case 语句

在这个示例中，您将明白 select 语句执行最先收到消息的接收者对应的 case 语句。

1. 打开本书代码示例中的 hour12/example06.go。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example06.go 运行这个程序。
4. 您将在终端中看到如下内容。

```
received ping on channel2
```

从这个示例可知，要根据最先收到的消息采取相应的措施，select 语句是一个不错的选择。但如果没有收到消息呢？为此可使用超时时间。这让 select 语句在指定时间后不再阻塞，以便接着往下执行。在程序清单 12.6 中，可添加一个超时 case 语句，指定在 0.5s 内没有收到消息时将采取的措施。

```
select {
    case msg1 := <-channel1:
        fmt.Println("received", msg1)
    case msg2 := <-channel2:
        fmt.Println("received", msg2)
    case <-time.After(500 * time.Millisecond):
        fmt.Println("no messages received. giving up.")
}
```

TRY IT YOURSELF

给 select 语句指定超时时间

在这个示例中，您将明白如何在 select 语句指定超时时间。

1. 打开本书代码示例中的 hour12/example07.go。

2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example07.go 运行这个程序。
4. 您将在终端中看到如下内容。

```
no messages received. giving up.
```

▲

12.6 退出通道

在已知需要停止执行的时间的情况下，使用超时时间是不错的选择，但在有些情况下，不确定 select 语句该在何时返回，因此不能使用定时器。在这种情况下，可使用退出通道。这种技术并非语言规范的组成部分，但可通过向通道发送消息来理解退出阻塞的 select 语句。

来看这样一种情形：程序需要使用 select 语句实现无限制地阻塞，但同时要求能够随时返回。通过在 select 语句中添加一个退出通道，可向退出通道发送消息来结束该语句，从而停止阻塞。可将退出通道视为阻塞式 select 语句的开关。对于退出通道，可随便命名，但通常将其命名为 stop 或 quit。在下面的示例中，在 for 循环中使用了一条 select 语句，这意味着它将无限制地阻塞，并不断地接收消息。通过向通道 stop 发送消息，可让 select 语句停止阻塞：从 for 循环中返回，并继续往下执行。

```
messages := make(chan string)
stop := make(chan bool)

for {
    select {
    case <-stop:
        return
    case msg := <-messages:
        fmt.Println(msg)
    }
}
```

在应用程序的某部分向通道发送消息，并要在未来的某个位置时点终止时，这种技术是很有效的。

为了提供这样的示例，我们在 Goroutine 中创建一个函数，它每隔 1s 向通道发送一条消息。

```
func sender(c chan string) {
    t := time.NewTicker(1 * time.Second)
    for {
        c <- "I'm sending a message"
        <-t.C
    }
}

messages := make(chan string)
go sender(messages)
```

通过在 for 循环中使用 select 语句，可在收到消息后立即打印它。由于这是一个阻塞操作，因此将不断打印消息，直到您手动终止这个过程。

```
for {
    select {
    case msg := <-messages:
        fmt.Println(msg)
    }
}
```

如果您要执行这个示例，可在本章的示例文件 example08.go 中找到它。

在这个示例中，除非杀死进程，否则不能退出这个程序——它将没完没了地运行下去。通过创建一个退出通道，可让程序向这个通道发送一条消息，从而结束 for 循环。

```
stop := make(chan bool)

for {
    select {
    case <- stop:
        return
    case msg := <-messages:
        fmt.Println(msg)
    }
}
```

程序清单 12.7 是一个完整的退出通道使用示例。在这个示例中，等待一定的时间后向退出通道发送了消息。但在实际工作中，具体等待多长时间可能取决于程序其他地方的未知事件何时发生。

程序清单 12.7 使用退出通道

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func sender(c chan string) {
 9:      t := time.NewTicker(1 * time.Second)
10:          for {
11:              c <- "I'm sending a message"
12:              <-t.C
13:      }
14:  }
15:
16:  func main() {
17:      messages := make(chan string)
18:      stop := make(chan bool)
19:      go sender(messages)
20:      go func() {
21:          time.Sleep(time.Second * 2)
22:          fmt.Println("Time's up!")
23:          stop <- true
24:      }()
25:
26:      for {
27:          select {
28:          case <-stop:
29:              return
30:          case msg := <-messages:
31:              fmt.Println(msg)
32:          }
33:      }
34:  }
```

▼TRY IT YOURSELF

使用退出通道

在这个示例中，您将明白如何使用退出通道。

1. 打开本书代码示例中的 hour12/example09.go。
2. 阅读这些代码，尝试理解它们是做什么的。
3. 在终端中使用命令 go run example09.go 运行这个程序。
4. 您将在终端中看到如下内容。

```
I'm sending a message
I'm sending a message
Time's up!
```

▲

12.7 小结

本章介绍了通道。通道与 Goroutine 一道提供了一种强大的并发性管理方式。您明白了如何创建通道以及如何使用它在 Goroutine 之间发送消息；您学习了如何创建缓冲通道以及如何将通道作为参数传递给函数；您了解到，可使用 select 语句来等待来自多个通道的消息，并根据来自哪个通道的消息最先到达来采取相应的措施。并发编程是一个庞大而复杂的主题，但只要掌握了如何使用通道在 Goroutine 之间通信，就能够实现很多并发功能！

12.8 问与答

问：可给通道执行多种数据类型吗？

答：不能。通道只能有一种数据类型。您可创建任何类型的通道，因此可使用结构体来存储复杂的数据结构。

问：在 select 语句中，如果同时从两个通道那里收到消息，结果将如何？

答：将随机地选择并执行一条 case 语句，且只执行被选中的 case 语句。

问：关闭通道时会导致缓冲的消息丢失吗？

答：关闭缓冲通道意味着不能再向它发送消息。缓冲的消息会被保留，可供接收者读取。

12.9 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

12.9.1 小测验

1. 相比于 Goroutine，通道有何优点？

2. select 语句中的超时时间有何用途？

3. 如何编写从一个通道那里接收 10 条消息后退出的程序？

12.9.2 答案

1. 通道与 Goroutine 互为补充，它让 Goroutine 能够相互通信。这给程序员提供了一个受控的并发编程环境。

2. 通过使用超时时间，可在指定时间过后从 select 语句返回，从而结束阻塞操作。select 语句根据最先到达的消息执行相应的 case 语句；通过指定超时时间，可在给定时间内没有收到任何消息时从 select 语句返回。

3. 要从一个通道那里接收 10 条消息后退出，可在迭代 10 次的 for 循环中使用 select 语句。

12.10 练习

修改介绍 Goroutine 的第 10 章的 example05.go，使其使用通道。在其中使用一个包含 select 语句的 for 循环，并指定超时时间，如果在指定时间内没有收到响应，则直接返回。本章示例代码中的 example10.go 提供了一种解决方案。

第 13 章

使用包实现代码重用

本章介绍如下内容。

- 导入包。
- 使用第三方包。
- 安装第三方包。
- 管理第三方依赖。
- 创建包。

在 Go 语言中，包用于将代码编组，以便在 Go 程序中导入并使用它们。本章介绍包以及如何使用它们来创建 Go 程序，还将介绍依赖管理以及如何创建并分享包。

13.1 导入包

为理解如何导入并使用包，我们来看使用 Go 语言编写的简单 Hello World!程序，如程序清单 13.1 所示。

程序清单 13.1 导入包

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 6:  )
 7:
 8:  func main() {
 9:      fmt.Println("Hello World!")
10:  }
```

Go 程序以 package 语句打头。main 包是一种特殊的包，其特殊之处在于不能导入。对 main 包的唯一要求是，必须声明一个 main 函数，这个函数不接受任何参数且不返回任何值。简而言之，main 包是程序的入口。

在 main 包中，可使用 import 声明来导入其他包。导入包后，就可使用其中被导出的（即公有的）标识符。在 Go 语言中，标识符可以是变量、常量、类型、函数或方法。这让包能够通过接口提供各种功能。例如，math 包提供了常量 Pi，如程序清单 13.2 所示。

程序清单 13.2　访问 math 包中的 Pi

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "math"
 6:  )
 7:
 8:  func main() {
 9:      fmt.Println(math.Pi)
10:  }
```

举一个函数导出的例子，strings 包导出了函数 ToLower，可用于将字符串转换为小写，如程序清单 13.3 所示。

程序清单 13.3　将字符串转换为小写

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 6:  )
 7:
 8:  func main() {
 9:      fmt.Println(strings.ToLower("STOP SHOUTING!"))
10:  }
```

导入包并使用其中导出的标识符，是重用标准库和其他第三方代码的基本方式。这种方式虽然简单，但您必须理解，灵活性和代码重用在很大程度上都是通过这种方式实现的。

13.2　理解包的用途

要了解如何使用包，先得知道该使用哪个包。在标准库中，Go 语言设计者采用了一致的命名约定，以帮助程序员明白包是如何组织的。在 Go 语言中，包名短小精悍又含义丰富。strings 包包含用于处理字符串的函数，bytes 包包含用于处理字节的函数。随着 Go 语言的使用经验越来越丰富，您就会知道哪些包是用于完成哪些任务的；如果实在记不起来，还可参阅直观而组织良好的文档。

假设您要在程序中操作字符串，那么通过阅读标准库包清单，您可能发现有一个 strings 包。您可在程序中导入并使用这个包！但如何知道它提供了哪些功能呢？Go 语言的文档很完善，而对于标准库中的包，都有卓越的文档。对于 strings 包，其文档可在 Go 语言官网找到。

在这个文档中，列出了所有被导出的标识符。假设您需要在程序中将字符串转换为小写，通过查看文档可知，有一个名为 ToLower 的函数。文档还指出这个函数将一个字符串作为参数，并返回一个字符串。

```
func ToLower(s string) string
```

文档简单地描述了这个函数的作用，还包含一些演示其用法的示例代码。对于这些示例代码，您可直接在浏览器中执行它们。如果您要更深入地了解这个函数，还可查看 strings 包的源代码。

13.3 使用第三方包

标准库提供了很多功能，但 Go 语言的设计理念是确保核心标准库小巧而稳定，因此标准库没有提供连接到数据库、分析文件格式以及实现身份验证协议的功能。不用多久，标准库就无法满足您编写程序的需求了。在这种情况下，程序员有两种选择。

- 自己编写解决问题的代码。
- 寻找能够解决问题的包（或库代码）。

大部分程序员很可能选择第二种做法。然而，在程序中添加额外的依赖要三思而行，因为这可能影响程序的稳定性和可维护性。考虑使用第三方库时，您应自问如下几个问题。

- 我明白了这些代码是做什么的吗？
- 这些代码值得信任吗？
- 这些代码的维护情况如何？
- 我真的需要这个库吗？

回答这些问题时，请考虑如下几点。

- 明白包的作用至关重要。优秀的第三方包都有卓越的文档，这些文档通常遵循 Go 语言文档约定，指出了它们导出了哪些标识符。通过阅读文档，可确定包是否提供了您所需的功能。
- 确定第三方包值得信任很重要。别忘了，将包导入程序后，它就能访问底层的操作系统。要确定第三方包的可信任程度，可了解还有多少人在使用它、是否有同事推荐，还可阅读其源代码。
- 考虑到软件的特征，第三方包不可避免地存在 Bug。不要选择几年都没有更新的包，而应选择开发方积极维护的第三方包，因为这意味着随着时间的推移，这样的包会越来越稳定。
- 导入第三方包会增加程序的复杂性。很多时候导入一个包只为了使用其中的一个函数，在这种情况下，可复制这个函数，而不导入整个包。

13.4 安装第三方包

要使用第三方库，必须像使用标准库一样使用 import 语句导入它。

在下面的示例中，将使用 Go 小组开发的 stringutil 包。这是一个简单的第三方包，只有一个函数被导出——Reverse。这个函数将一个字符串作为参数，将该字符串反转并返回结果。

要使用第三方包，必须先使用命令 go get 安装它。这个命令默认随 Go 一起安装了，它将指向远程服务器中包的路径作为参数，并在本地安装指定的包。

```
go get github.com/golang/example/stringutil
```

这个包被安装到环境变量 GOPATH 指定的路径中，因此可在程序中使用它。要查看这个包的源代码，可打开目录 src 中的文件。包的安装目录如下。

```
// OSX and Linux
$GOPATH/src/github/golang/example/stringutil
// Windows
%GOPATH%\src\github\golang\example\stringutil
```

安装这个包后，就可导入它了，代码如程序清单 13.4 所示。

程序清单 13.4 使用第三方包

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "github.com/golang/example/stringutil"
 6:  )
 7:
 8:  func main() {
 9:      s := "ti esrever dna ti pilf nwod gniht ym tup I"
10:      fmt.Println(stringutil.Reverse(s))
11:  }
```

如果您运行这个使用第三方包的程序，它将反转字符串。

```
go run example01.go
I put my thing down flip it and reverse it
```

通常，第三方包依赖于其他第三方包。命令 go get 很聪明，它会下载依赖的第三方包，让您无须手工安装每个包依赖的第三方包。

TRY IT YOURSELF ▼

安装并使用第三方包

在这个示例中，您将明白如何安装并使用第三方包。

1. 在文本编辑器中打开文件 hour13/example01.go，并尝试理解这个示例是做什么的。
2. 使用命令 go get github.com/golang/example/stringutil 安装 stringutil 库。
3. 使用命令 go run example01.go 运行这个示例。
4. 您将看到反转后的文本。

```
I put my thing down flip it and reverse it
```

▲

13.5 管理第三方依赖

很多语言都有包管理器，可简化使用第三方包的工作：Python 有 pip、.NET 有 Nuget、

Ruby 有 RubyGems、Node.js 有 npm。本书编写期间，Go 还没有官方包管理器，但 dep 正在开发过程中。

本章前面介绍了如何安装远程包，但使用第三方包时，还需考虑众多不同的因素。

- 如何更新包以修复缺陷？
- 如何指定版本？
- 如何与其他开发人员分享依赖清单？
- 如何在构建服务器中安装依赖？

使用命令 go get 可更新文件系统中特定的包或所有的包。要更新项目的依赖，可在项目文件夹中执行如下命令。

```
go get -u
```

也可只更新特定的包。

```
go get -u github.com/spf13/hugo
```

还可更新文件系统中所有的包。

```
go get -u all
```

命令 go get 从与本地分支匹配的远程分支中获取源代码，例如，如果本地分支为 master，则这个命令将从远程分支 master 获取最新的源代码。

在大多数情况下，更新包都很简单，但有时可能很复杂，例如，在多个项目中使用了同一个第三方库时：项目 A 依赖于某个第三方库的 1.2 版，而项目 B 依赖于 1.3 版。

为应对这种情况，Go 1.5 版引入了文件夹 vendor，这能够让您将第三方模块添加到项目目录下的文件夹 vendor 中，并将所有包文件都移到这个文件夹中。这样可以不全局地安装包，而仅在项目中安装它。对于前面安装的 stringutil 包，可将其移到文件夹 vendor 中，这样项目结构将如下。

```
example02/
├── example02.go
└── vendor
    └── github.com
        └── golang
            └── example
                └── stringutil
                    ├── reverse.go
                    └── reverse_test.go
```

请注意，现在将只能在这个项目中使用 stringutil 包，因为它不是全局的。这种做法有一些优点。

- 可锁定包的版本，为此只需将特定版本复制到项目目录中。
- 构建服务器无须下载依赖，因为它们包含在项目中。

如果您使用过包管理器，可能会发现这种做法的一些缺点。

- 依赖必须包含在仓库中。

- 使用的是包的哪个版本不明显。
- 没有处理包的依赖。
- 无法在清单文件中准确地指定提交或分支。

当前，程序员必须手工管理复杂的依赖，有很多第三方工具使用 vendor 文件夹支持安装特定的包版本。

TRY IT YOURSELF ▼

使用文件夹 vendor

在这个示例中，您将明白如何使用文件夹 vendor 来管理依赖。

1. 这个示例要求将本书的代码示例放在环境变量 GOPATH 指定的路径中。如果您没有使用 go get 安装这些代码示例，现在就这样做：go get github.com/shapeshed/golang-book-examples。
2. 打开本书代码示例中的 hour13/example02。
3. 查看文件夹 vendor，它是 stringutil 包的本地副本。
4. 就这个示例而言，无须执行命令 go get。
5. 使用命令 go run example02.go 运行这个程序。
6. 您将发现字符串被反转。

```
I put my thing down flip it and reverse it
```

▲

13.6 创建包

除使用第三方包外，有时还可能需要创建包。本节将创建一个示例包，并将其发布到 Github 以便与人分享。这是一个处理温度的包，提供了在不同温度格式之间进行转换的函数。请创建一个名为 temperature.go 的文件，并在其中添加如程序清单 13.5 所示的内容。

程序清单 13.5 一个简单的包

```
1:  package temperature
2:
3:  func CtoF(c float64) float64 {
4:      return (c * (9 / 5)) + 32
5:  )
6:
7:  func FtoC(c float64) float64 {
8:      return (f-32) * (9 / 5)
9:  }
```

别忘了，导入这个包后，就可使用其中所有以大写字母打头的标识符了。要创建私有标识符（变量、函数等），可让它们以小写字母打头。

为测试这个包，可创建一个测试文件，并将其命名为temperature_test.go，如程序清单13.6所示。测试将在第15章介绍，就目前而言，只需知道这个文件对这个包进行测试就行了。

程序清单13.6 对包进行测试

```
 1:  package temperature
 2:
 3:  import (
 4:      "testing"
 5:  )
 6:
 7:  type temperatureTest struct {
 8:      i float64
 9:      expected Temperature
10:  }
11:
12:  var CtoFTests = []temperatureTest{
13:      {4.1, 39.38},
14:      {10, 50},
15:      {-10, 14},
16:  }
17:
18:  var FtoCTests = []temperatureTest{
19:      {32, 0},
20:      {50, 10},
21:      {5, -15},
22:  }
23:
24:  func TestCtoF(t *testing.T) {
25:      for _, tt := range CtoFTests {
26:          actual := CtoF(tt.i)
27:          if actual != tt.expected {
28:              t.Errorf("expected %v, actual %v", tt.expected, actual)
29:          }
30:      }
31:  }
32:
33:  func TestFtoC(t *testing.T) {
34:      for _, tt := range FtoCTests {
35:          actual := FtoC(tt.i)
36:          if actual != tt.expected {
37:              t.Errorf("expected %v, actual %v", tt.expected, actual)
38:          }
39:      }
40:  }
```

如果您运行这些测试，将发现它们都通过了。

```
go test
PASS
ok      github.com/shapeshed/temperature      0.001s
```

对于要发布到网上的包，从用户的角度考虑问题很重要。因此，推荐在包中包含如下3个文件。

- 指出用户如何使用代码的LICENSE文件。
- 包含有关包的说明信息的README文件。
- 详细说明包经过了哪些修改的Changelog文件。

作为代码的创建者，如何授权由您决定。有很多开源许可方式，有些要求宽松，有些要

求严格。

在 README 文件中，应包含有关包的信息、如何安装包以及如何使用包。您可能还想在其中包含有关如何参与改进项目的信息。如果要将包发布到 Github，应考虑使用 Markdown 格式编写，因为 Github 会自动设置 markdown 文件的格式。

在 Changelog 文件中，应列出对包所做的修改，这可能包含功能添加情况和 API 删除情况。通常，使用 git 标签来指示发布情况，能够让用户轻松地下载特定版本。

13.7 小结

本章介绍了包，这是一种封装、重用和分享 Go 代码的方式。您学习了如何安装包以及如何在 Go 程序中使用它们；您学习了如何使用文件夹 vendor 来管理依赖，明白了如何编写自定义包；最后，您了解到可通过网络分享自己创建的包。

13.8 问与答

问：在什么情况下应自己编写代码？在什么情况下应使用第三方包？

答：这个问题很难回答！对于初学者来说，使用第三方包很有吸引力，而且通常效率更高。随着时间的推移，您将对 Go 语言越来越熟悉，对第三方包的依赖越来越少。对于诸如数据库驱动程序等专用代码，使用第三方包是合理的选择，但在其他情况下，自己动手编写代码或复制第三方包的一部分可能才是合理的选择。

问：应在多大程度上关注安全风险？

答：在项目中引入任何第三方代码都存在安全风险。Go 语言没有提供包签名，因此攻击者可能破解 Github 账户，在流行的包中植入恶意代码。实际上，发生这种情况的可能性不大，需要多小心取决于您对安全的要求。例如，大学项目和军事项目对安全的要求不可相提并论。

问：如何将第三方包的依赖复制到文件夹 vendor 中？

答：编写本书期间，Go 没有提供将第三方包的依赖移到文件夹 vendor 的官方工具。您需要手工完成这种任务，或使用第三方依赖管理工具。

13.9 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

13.9.1 小测验

1. 在包文件中，以大写字母打头的标识符和以小写字母打头的标识符有何不同？
2. 如何安装第三方包？

3．文件夹 vendor 有何作用？

13.9.2 答案

1．以大写字母打头的标识符会被导出，这意味着导入包后就可使用它们；以小写字母打头的标识符不会被导出，这意味着即便导入包也无法使用它们。简而言之，以大写字母打头的标识符是公有的，而以小写字母打头的标识符是私有的。

2．要安装第三方包，可使用命令 go get 并指定相应的域名和路径。包将安装到环境变量 GOPATH 指定的路径中，并可在位于环境变量 GOPATH 指定路径的程序中使用它们。

3．文件夹 vendor 能够让 Go 将位于该文件夹中的包链接到程序，而不使用全局安装的版本。这意味着可准确地指定要使用哪个版本的包。有几个第三方工具还使用这个文件夹实现了其他功能。

13.10 练习

1．请阅读 strings 包的源代码，并确定哪些标识符被导出。如果您不能完全看懂这些代码，这不用担心，只要理解包的工作原理就行。

2．在网上查找用于读取和分析 toml 格式文件的第三方包。通过阅读其项目主页，您能够理解这个包的用途和用法吗？请安装这个包，并创建一个使用它的简单程序，以证明您确实知道如何使用它。

3．对本章创建的 temperature 包进行扩展，提供一个将摄氏温度转换为绝对温度的函数。添加一个表格驱动测试（table test），对一系列值进行测试。

第 14 章

Go 语言命名约定

本章介绍如下内容。

- Go 代码格式设置。
- 使用 **gofmt**。
- 配置文本编辑器。
- 命令约定。
- 使用 **golint**。
- 使用 **godoc**。
- 工作流程自动化。

如果您去探索 Go 语言生态系统，应该会经常遇到“Go 语言惯常方式”（Idiomatic Go）一词，它指的是被普遍接受的行事方式。对于“Go 语言惯常方式”的含义，没有强制的标准，也没有相关的编译器检查，因此在有些情况下，“Go 语言惯常方式”可能显得神秘而晦涩。本章探讨 Go 语言约定以及有助于遵循“Go 语言惯常方式”的工具。实际上，“Go 语言惯常方式”意味着遵循接下来将介绍的标准约定。阅读本书后，您将明白 Go 社区遵循的众多约定。

14.1 Go 代码格式设置

代码格式设置指的是如何在文件中设置代码的格式。具体地说，它指的是如何缩进代码以及如何使用回车。在代码格式设置方面，Go 语言没有强制的约定，但存在被整个 Go 社区广泛采用并遵循的事实标准。请看程序清单 14.1，这些 Go 代码完全合法，能够编译并执行，但您觉得它们易于阅读吗？

程序清单 14.1 Go 代码格式设置

```
1:  package main        ; import "fmt";
2:  func main() { fmt.Println("Hello World")}
```

代码格式设置风格是一个常常在程序员之间引起争论的主题，在大多数情况下，这些争论都让人专注于格式而不是代码。另外，对使用不同风格编写的代码库进行维护时，可能导致代码晦涩难懂、在合并请求方面出现争议甚至引入错误。例如，JavaScript 在语法方面非常宽松，常常在该如何编写代码方面引发争议。在 Node.js 中，下述导入模块的方式都合法。

```
var http = require("http");
var crypto = require("crypto");

var http = require("http")
  , crypto = require("crypto")

var http = require("http"),
    crypto = require("crypto");
```

虽然 Node.js 社区进行了一些标准化，但经常能在项目中同时看到这 3 种风格。在很多情况下，给开源项目贡献的代码都没有遵循大家普遍接受的约定，需要重新设置它们的格式。可能令人难以置信的是，这种与代码无关的问题常常会给项目带来麻烦。

在代码格式设置方面，Go 语言采取了实用而严格的态度。Go 语言指定了格式设置约定，这种约定虽然并非强制性的，但命令 gofmt 可以实现它。虽然编译器不要求按命令 gofmt 指定的那样设置代码格式，但几乎整个 Go 社区都使用 gofmt，并希望按这个命令指定的方式设置代码格式。

对于这种格式设置方面的默认选择，有些程序员最初很不满意，但实际上，在官方代码中做出这样的选择，对大家来说意味着解放，因为这样开发小组就无须花时间去讨论哪种风格是正确的，进而制定风格指南了。强烈建议您按 Go 语言约定设置代码的格式。

14.2 使用 gofmt

为确保按要求的约定设置 Go 代码的格式，Go 提供了 gofmt。这个工具的优点在于，让您甚至都无须了解代码格式设置约定。通过不断地学习如何设置代码格式，您自然而然地就会遵循代码格式设置约定。对于前面的程序清单 14.1，可使用工具 gofmt 来设置其格式，使其遵循代码格式设置约定。对文件运行 gofmt 时，将把结果打印到标准输出，但不修改原来的文件。

```
gofmt example01.go
package main

import "fmt"

func main() {
    fmt.Println("Hello World")
}
```

请注意，这个文件的格式被重新设置了，这使其更容易阅读且间隔是一致的。要确定文件与要求的约定有何不同，可使用选项-d 来显示差别。

```
diff example01.go gofmt/example01.go
--- /tmp/gofmt062251244 2017-08-26 14:38:12.554082827 +0100
+++ /tmp/gofmt593374555 2017-08-26 14:38:12.554082827 +0100
```

```
@@ -1,4 +1,7 @@
-package main; import "fmt";
+package main

-func main() {
-fmt.Println("Hello World") }
+import "fmt"
+
+func main() {
+        fmt.Println("Hello World")
+}
```

要让工具 gofmt 修改文件，可使用标志 -w。这将使设置格式后的结果覆盖当前文件。

```
gofmt -w example01.go
```

TRY IT YOURSELF ▼

使用 gofmt

在这个示例中，您将学会如何使用工具 gofmt。

1. 打开本书代码示例中的 hour14/example01.go。
2. 注意到其中的代码格式设置很糟。
3. 在终端中执行命令 gofmt -w example01.go。
4. 打开文件 example01.go，工具 gofmt 重新设置了代码的格式。

▲

14.3 配置文本编辑器

很多文本编辑器都有第三方 Go 插件，它们在您保存文件时自动运行 gofmt 以校正格式设置。在这些工具中，很多都会正确地配置文本编辑器，使其像 Go 语言约定要求的那样使用制表符而不是空格来缩进。通过在文本编辑器中使用插件，可自动使用诸如 gofmt 等工具。

- Vim(vim-go)
- Emacs(go.mode.el)
- Sublime(GoSublime)
- Atom(go-plus)
- Eclipse(goclipse)
- Visual Studio(vscode-go)

在这些插件中，很多不仅支持简单的格式设置，还能够让开发人员直接在文本编辑器中构建、测试和运行代码。虽然编写 Go 代码时并非必须使用这些插件，但使用它们可提高效率，因此建议了解有哪些 Go 插件可用于您使用的文本编辑器。

14.4 命名约定

据说在计算机科学中，最难的两件事是缓存和命名。虽然这种说话不能当真，但清晰的命名可提高代码的可读性和可维护性。在 Go 语言中，有些约定是编译器要求必须遵循的，而有些是否遵循由程序员决定。像其他程序员那样遵循 Go 语言约定有助于您成为 Go 社区中的良好公民。就命名而言，第 13 章说过，以大写字母打头的标识符将被导出，而以小写字母打头的不会。

```
var Foo := "bar" // Exported
var foo := "bar" // Not Exported
```

很多其他语言也有有关公有和私有变量方面的约定，如使用下划线表示私有变量。在 Go 语言中，不应使用这些约定，而应遵循大小写约定。

就如何给变量命名方面，所有程序员达成一致的可能性不大，但只要采用一致的方式，这个答案就不那么重要了。在 Go 语言中，对于包含多个单词的变量名，约定是使用骆驼拼写法或帕斯卡拼写法，具体使用哪种拼写法，取决于变量是否需要导出。

```
var fileName // Camel Case
var FileName // Pascal Case
```

在 Go 程序中，经常使用指出了数据类型的简短变量名，这让程序员能够专注于逻辑而不是变量。在这种情况下，i 表示整型（Integer）数据类型，s 表示字符串数据类型，等等。一开始，您可能觉得这样做不好，但时间久了就会习惯这种被普遍接受的约定。

```
var i int = 3
var s string = "hello"
var b bool = true
```

良好的命名还可提高代码的可读性。第 4 章介绍了函数签名，而第 8 章介绍了方法。等您熟悉函数签名的工作原理后，使用良好的方式给函数和方法命名可让其代码的含义不言自明。请看下面两个函数签名。

```
func a(f float64) float64
func (t *Triangle) Area() float64
```

在第一个函数签名中，函数和变量名更简洁，但仅根据函数签名难以确定函数是做什么的。第二个是一个方法，给方法和接收者参数都指定了贴切的名称。显然，这个方法是用于三角形的！方法名也明确地指出了它是做什么的。只需通过接收者参数和方法的名称，您就立即能够知道它计算三角形的面积。

第 8 章介绍了接口，它们是具名的方法签名集合。在 Go 源代码中，接口名通常是这样得到的：在动词后面加上后缀 er，形成一个名词。后缀 er 通常表示操作，因此这种命名方式表示操作，如 Reader、Writer 和 ByteWriter。在有些情况下，这样生成的接口名可能不是现成的英语单词，如果您在 Go 源代码中搜索，将找到诸如 Twoer 这样的接口名。

对于要导出的函数，命名约定是尊重这样的事实，即导入包后，就可通过包名和函数名

来访问它。例如，在标准库中，math 包就遵守了这样的约定：将计算平方根的函数命名为 Sqrt 而不是 MathSqrt。这合乎情理，因为使用这个函数时，代码为 math.Sqrt 而不是 math.MathSqrt，另外，只要通过这个函数名就能知道它是做什么的。即便不查看函数的实现，程序员也能轻易知道它是做什么的。程序清单 14.2 使用了 math 包来计算平方根。Go 设计者遵循的约定让这些代码很容易理解。

程序清单 14.2 使用 math 包计算平方根

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5:     "math"
 6: )
 7:
 8: func main() {
 9:     var f float64 = 9
10:     fmt.Println(math.Sqrt(f))
11: }
```

命名总是带有一定的主观性，但花点时间考虑如何命名总是值得的。给变量、函数和接口命名时，需要考虑的一些因素包括以下几点。

- 谁将使用这些代码？只是我自己还是整个团队？
- 是否为当前项目制定了命名约定？
- 不熟悉代码的人是否只需看一眼就能大致知道它是做什么的？

遵循命名约定很重要，但过于教条也是有害的。您需要考虑代码所在的上下文、其他相关人员以及小组的稳定性。在大多数情况下，都应该兼顾约定以及代码所在的上下文。

14.5 使用 golint

golint 是 Go 语言提供的一个官方工具。gofmt 根据指定的约定设置代码的格式，而命令 golint 根据 Go 项目本身的约定查找风格方面的错误。默认不会安装 golint，但可像下面这样安装它。

```
go get -u github.com/golang/lint/golint
```

要核实是否正确地安装了这个工具，可在终端中执行命令 golint --help。如果正确地安装了它，您将在终端中看到一些帮助文本。工具 golint 提供了有关风格方面的提示，还可帮助学习 Go 生态系统广泛接受的约定。程序清单 14.3 是一些在风格方面有待改进的 Go 代码。

程序清单 14.3 在风格方面有待改进的 Go 代码

```
 1: package main
 2:
 3: import "fmt"
 4:
 5: const Foo string = "constant string"
 6:
 7: func main() {
 8:     fmt.Println(Foo)
 9:     a_string := "hello"
```

```
10:       fmt.Println(a_string)
11:   }
```

这些代码能够通过编译，也能通过 gofmt 的检查。能够通过编译说明这些代码正确无误，能够通过 gofmt 的检查说明格式也没有问题。但您可能意识到了，这些代码存在一些风格方面的问题。例如，您可能注意到，有个变量名包含下划线。使用工具 golint 可找出这种风格方面的问题。如果对这些代码运行 golint，它将提供一些建议，指出从风格方面改进这些代码。

```
go lint example02.go
example02.go:5:7: exported const Foo should have comment or be unexported
example02.go:9:2: don't use underscores in Go names; var a_string should be aString
```

golint 可能指出代码的哪些地方（行号和位置）需要注意。如果您使用的文本编辑器是 Vim，这些信息还将集成到快速修复菜单中。这些改进建议不是强制性的，因为代码能够通过编译，但鉴于这个 Go 项目使用了这个工具，所以最好尽力消除这些警告并学习相关的约定。使用工具 golint 还是一种学习以 Go 语言惯用方式编写代码的绝妙途径。工具 golint 涵盖了很多约定，包括命名、风格和一般性约定，建议在您的 Go 项目中使用它。很多文本编辑器插件都让您能够在保存项目时运行 golint，对项目运行测试时您可能应该考虑这样做，或通过定期地运行 golint 来强化学习。

▼ TRY IT YOURSELF

理解 golint

在这个示例中，您将学习如何使用工具 golint。

1. 打开本书代码示例中的 hour14/example02.go。
2. 您将发现其中有些地方在风格方面有待改进。
3. 在终端中执行命令 golint example02.go。
4. 您将在终端中看到如下内容。

```
go lint example02.go
example02.go:5:7: exported const Foo should have comment or be unexported
example02.go:9:2: don't use underscores in Go names; var a_string should be
aString
```

▲

14.6 使用 godoc

随着要开发的程序越来越复杂，要确保其品质优良，编写文档至关重要。即便您单独开发，注释也有助于您快速理解代码的作用。在文档编写方面，Go 语言提供了良好的支持；在确保文档编写工作尽可能简单方面，Go 语言的设计者做了深入的考虑。

godoc 是一款官方工具，可通过分析 Go 语言源代码及其中的注释来生成文档。由于文档是根据源代码生成的，这很大程度上避免了文档不同步（软件项目中常见的问题）的问题。

虽然 godoc 是一款官方工具，但它必须单独安装。如果 Go 环境已搭建好，要安装这款

工具只需安装相应的包即可。

```
go get golang.org/x/tools/cmd/godoc
```

要核实是否正确地安装了这款工具，可在终端中执行命令 godoc--help。如果安装正确，您将在终端中看到一些帮助文本。

如果使用过其他编程语言，那么您可能熟悉代码注释约定。在有些情况下，这些约定要求您以特定的方式编写注释，以便能够根据它们生成文档。Java 提供了根据注释生成文档的 javadoc，程序清单 14.4 使用了一些文档注释来说明代码。

程序清单 14.4 javadoc

```
/**
 * The HelloWorld program just says hello
 *
 * @author George Ornbo
 * @version 1.0
 * @since 2017-08-17
 */
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

在这些 Java 代码中，有些字段前面有@，让 javadoc 知道去哪里查找特定的信息，如作者和版本号。工具 godoc 不要求这样做，您只需使用标准注释对代码进行注释，并遵守一些简单约定即可。要给一段代码添加注释，只需在注释行开头指出要注释的元素的名称。程序清单 14.5 是一个示例包，其中包含一些根据 godoc 约定添加的注释。

程序清单 14.5 使用 godoc

```
// Package example03 shows how to use the godoc tool.
package example03

import (
    "errors"
)

// Animal specifies an animal
type Animal struct {
    Name string // Name holds the name of a thing.

    // Age holds the name of a thing.
    Age int
}

// ErrNotAnAnimal is returned if the name field of the Animal struct is Human.
var ErrNotAnAnimal = errors.New("Name is not an animal")

// Hello sends a greeting to the animal.
func (a Animal) Hello() (string, error) {
    if a.Name == "Human" {
        return "", ErrNotAnAnimal
    }
    s := "Hello " + a.Name
    return s, nil
}
```

考虑到本章的目标，这个示例中的代码不如给它们添加注释的方式重要。请注意，每行都以它注释的类型的名称打头。注释是以大写字母打头的完整句子，以点结束。如果对这些代码运行工具 godoc，将生成有关这个包的文档。

```
godoc ./example03
PACKAGE DOCUMENTATION

package example03
    import "./example03"

    Package example03 shows how to use the godoc tool.

VARIABLES

var ErrNotAnAnimal = errors.New("Name is not an animal")
    ErrNotAnAnimal is returned if the name field of the Animal struct is
    Human.

TYPES

type Animal struct {
    Name string // Name holds the name of an Animal.

    // Age holds the name of an Animal.
    Age int
}
Animal specifies an animal

func (a Animal) Hello() (string, error)
    Hello sends a greeting to the animal.
```

▼ TRY IT YOURSELF

使用 godoc

在这个示例中，您将明白如何使用工具 godoc。

1. 打开本书代码示例中的 hour14/example03.go。
2. 注意到在代码中添加了一些注释。
3. 在终端中运行命令 godoc example03.go。
4. 您将在终端中看到生成的文档。

▲

工具 godoc 能够生成很多不同的输出，其中包括 html。标准库文档就是使用 godoc 生成的，而很多第三方网站都提供了有关开源项目的 HTML 格式文档。遵循 godoc 指定的约定意味着可以通过标准方式生成文档，这样生成和维护代码文档将易如反掌。

要学习如何编写文档，一种绝妙的方式是研究标准库编写文档的做法。安装 godoc 后，就可将任何标准库的文档输出到终端中。例如，要查看 strings 包的文档，可执行如下命令。

```
godoc strings
```

这将把 strings 包的文档打印到标准输出中。如果您使用的是 UNIX 系统，可结合使用 grep 快速获悉要使用的方法的函数签名。

```
godoc string | grep "func Replace"
func Replace(s, old, new string, n int) string
```

另外，您还可通过启动一个 Web 服务器来查看标准库文档。在您没有连接到网络或连接速度有限时，这种做法是一个不错的选择。

```
godoc -http=":6060"
```

执行这个命令后，您就可在浏览器中输入地址 http://localhost:6060/pkg/来查看标准库文档。

TRY IT YOURSELF ▼

使用 godoc 读取文档

在这个示例中，您将明白如何使用 godoc 来读取文档。

1. 在终端中执行命令 godoc -http=":6060".
2. 打开 Web 浏览器，并输入地址 http://localhost:6060/pkg/。
3. 您将看到 Go 语言文档，并能够在其中导航。

▲

对标准库来说，以 godoc 要求的方式编写文档的好处显而易见，但这也适用于第三方包。很多第三方网站都提供了使用工具 godoc 生成的 HTML 格式的文档；如果您无法连接到网络，也可使用工具 godoc 来查看第三方项目的文档。github.com/BurntSushi/toml 包被广泛用于分析 TOML 格式数据，下面的示例演示了如何使用工具 godoc 来查看这个包的文档。

```
godoc $GOPATH/src/github.com/BurntSushi/toml
```

虽然开发人员可使用自己的约定，但 godoc 是事实标准，它提供了轻松编写和读取文档的标准方式。

14.7 工作流程自动化

本章介绍了如何使用 gofmt、golint 和 godoc，这些工具都很方便，但要随时记得运行它们可能是很困难的。即便您熟悉并知道如何使用这些工具，有时也可能忘记它们。鉴于此，推荐尽可能将执行代码检查工具的工作自动化。

根据您使用的文本编辑器，可能会有能够按需或在存盘时运行工具的插件。如果您使用的是 UNIX 操作系统，还可使用 Makefile。Makefile 可简化源代码编译、测试和检查工作，能够将多个名称组合起来，因此它是很有用的。另外，在持续集成环境中可使用它们来自动测试、检查和编译代码。

程序清单 14.6 显示了一个 Makefile，它对当前文件夹中的所有文件运行 gofmt。如果发

现有文件的格式设置不正确，就将其路径打印到终端。

程序清单 14.6 使用 Makefile

```
1:  all:  check-gofmt
2:
3:  check-gofmt:
4:  @if [ -n "$(shell gofmt -l .)" ]; then \
5:      echo 1>&2 'The following files need to be formatted:'; \
6:      gofmt -l .; \
7:      exit 1; \
8:  fi
```

要运行这个脚本，只需执行命令 make。如果当前文件夹中有格式设置不正确的文件，将把它们打印到终端。程序清单 14.1 列出了一些格式设置糟糕的代码，通过使用 Makefile，可根据 gofmt 指定的约定检查这个文件。Makefile 可实现专业的开发流程，因为使用它们可在将代码提交到仓库时自动执行检查。在这里，运行 make 时发现文件的格式设置不正确，因此生成相应的报告。

```
Make
The following files need to be formatted:
example04.go
make:  *** [Makefile:4: check-gofmt] Error 1
```

TRY IT YOURSELF

使用 makefile 实现工作流程自动化

在这个示例中，您将明白如何使用 Makefile 来实现工作流程自动化。

1. 打开本书代码示例中的文件夹 hour14/example04。
2. 在终端中执行命令 make。
3. 您将在终端中看到一条消息，指出文件的格式设置不正确。

```
go lint example02.go
example02.go:5:7: exported const Foo should have comment or be unexported
example02.go:9:2: don't use underscores in Go names; var a_string should be
aString
```

14.8 小结

本章介绍了 Go 语言约定和官方工具。虽然这些对编写 Go 代码来说并非必不可少，但它们提供了卓越的代码编写和分享环境。推荐您遵循这些官方工具指定的约定，这些工具包括 gofmt（用于设置 Go 文件的格式）、golint（提供有关风格和约定方面的建议）和 godoc（用于生成和读取文档）。通过遵守 Go 语言约定，就不用在风格和不同看法上浪费时间，而只需关心重要的事情——编写代码。

14.9 问与答

问：gofmt 在代码中添加的缩进量很大，我不喜欢，能够修改吗？

答：虽然您可能不喜欢某些约定，但约定的优点在于大家都遵循。随着时间的推移，您会将这种不喜欢忘在脑后的。

问：我习惯了将常量名全大写，如 CONSTANT，在 Go 语言中，也应这样做吗？

答：在 Go 语言中，常量与其他数据元素没什么不同，因此不应将其全大写。

问：对于我开发的开源 Go 项目，可将其文档托管到什么地方？

答：有很多第三方网站能够分析 Go 代码并生成 HTML 格式的文档。一个这样的流程网站是 GoDoc，有关如何将包上传到这里，请参阅 About 页面。

14.10 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

14.10.1 小测验

1．没有遵循 gofmt 指定的约定的代码能够运行吗？

2．为何说编写文档是好事一件？

3．在给接口命名方面，Go 语言惯用的做法是什么？

14.10.2 答案

1．只要代码能够通过编译，即便没有根据 gofmt 指定的约定设置格式，它也能够运行。

2．文档可提高代码的可读性和可维护性，让开发人员在重温自己编写的代码时能够迅速看懂，还让团队的其他人员能够更快地理解这些代码。

3．在接口命名方面，惯用的做法是使用一个动词，并在末尾加上 er，如 Parser 和 Authorizer，这样的名称描述了接口是做什么的。

14.11 练习

1．研究您使用的文本编辑器是否有 Go 插件。如果有，就安装它并看看它提供了哪些功能。

2．编写一个简单的 Hello World 程序，并给它编写文档，再使用工具 godoc 将这些文档打印到终端中。

3．在网上查找一个流行的开源 Go 项目，并下载其源代码，再对这些代码运行本章介绍的工具。

第 15 章

测试和性能

本章介绍如下内容。

- 测试的重要性。
- **testing** 包。
- 基准测试。
- 提供测试覆盖率。

本章介绍测试的重要性，您将为 Go 程序编写测试、明白测试为何很重要并学习可运行的各种测试，还将学习多种很有用的 Go 基准测试模式。阅读完本章后，您将能够对 Go 代码进行测试和基准测试。

15.1 测试：软件开发最重要的方面

测试软件程序可能是软件开发人员能够做的最重要的事情。通过测试代码的功能，开发人员能够在很大程度上确定程序是有效的。另外，每次修改代码后，开发人员都可运行测试，确认没有引入 Bug 和衰退。通过测试软件，还能够让软件工程师确认程序按期望的方式工作。

通常，软件测试是从概述功能的用户故事或规范衍生而来的。例如，如果有一个用户故事，指出一个函数接受两个数字，将它们相加并返回结果，就可轻松地编写对此进行检查的测试。有些项目还要求新代码有配套的测试。

编写良好的测试可充当文档。鉴于测试描述了程序期望的运行方式，新加入项目的开发人员通常可通过阅读测试来了解程序的运行方式。

常用的测试有多种。

- 单元测试。
- 功能测试。
- 集成测试。

15.1.1 单元测试

单元测试针对一小部分代码，并独立地对它们进行测试。通常，这一小部分代码可能是单个函数，而要测试的是其输入和输出。典型的单元测试可能指出，如果给函数 x 提供这些值，它应返回这个值。在确认程序最小的构件按期望的方式运行方面，这种测试很有用。在程序增大和变化过程中，单元测试是发现衰退的绝佳方式。衰退是修改过程中引入的 Bug 或错误。

衰退意味着代码在修改前有效，但修改后无效了。单元测试通常能够发现衰退，因为它们测试的是程序的最小组成部分。

15.1.2 集成测试

集成测试通常测试的是应用程序各部分协同工作的情况。如果说单元测试检查的是程序的最小组成部分，那么集成测试检查的就是应用程序各个组件协同工作的情况。集成测试还检查诸如网络调用和数据库连接等方面，以确保整个系统按期望的那样工作。通常，集成测试比单元测试更难编写，因为这些测试需要评估应用程序依赖的各个部分。

15.1.3 功能测试

功能测试通常被称为端到端测试或由外向内的测试。这些测试从最终用户的角度核实软件按期望的那样工作，它们评估从外部看到的程序的运行情况，而不关心软件内部的工作原理。对用户来说，功能测试可能是最重要的测试。下面是一些功能测试的例子。

- 测试命令行工具，确定在用户提供特定的输入时，它将显示特定的输出。
- 对网页运行自动化测试。
- 对 API 运行从外到内的测试，并检查响应代码和报头。

15.1.4 测试驱动开发

很多开发人员都提倡采用测试驱动开发（TDD）。这种做法从测试的角度考虑新功能，先编写测试来描述代码片段的功能，再着手编写代码。这很有多优点。

- 有助于描述代码设计，因为考虑清楚代码片段的工作原理后，可改善代码设计。
- 有助于提供有关功能工作原理的定义。
- 未来可使用现成的测试来确定没有发生衰退。
- 可使用现成的测试来核实正确地实现了代码。

通过采用 TDD，工程师可改善设计，并根据确保测试得以通过来确认代码是有效的。

15.2 testing 包

为支持测试，Go 语言在标准库中提供了 testing 包，它还支持命令 go。与 Go 语言的其

他众多方面一样，您也许理解一些与 testing 包相关的设计良好的约定。

第一个约定是，Go 测试与其测试的代码在一起。测试不是放在独立的测试目录中，而是与它们要测试的代码放在同一个目录中。测试文件是这样命名的：在要测试的文件的名称后面加上后缀 _test，因此如果要测试的文件名为 strings.go，则测试它的文件将名为 strings_test.go，并位于文件 strings.go 所在的目录中。

```
Project
 ├── strings.go
 └── strings_test.go
```

第二个约定是，测试为名称以单词 Test 打头的函数。程序清单 15.1 显示了一个测试，它检查布尔值 true 与它自身是否相等。如果连这种测试都不能通过，我们就有大麻烦了！

程序清单 15.1 运行基本测试

```
1:  package example01
2:
3:  import "testing"
4:
5:  func TestTruth(t *testing.T) {
6:      if true != true {
7:          t.Fatal("The world is crumbling")
8:      }
9:  }
```

对程序清单 15.1 解读如下。

- 导入了 testing 包。
- 函数名 TestTruth 表明这是一个测试，因为它以单词 Test 打头。
- 向这个函数传递了类型 T，它包含很多用于测试代码的函数。
- 使用 if 语句判断 true 是否与它自身相等。
- 这条 if 语句的结果为 true，因此测试将通过。

如果您运行程序清单 15.1，将发现这个测试通过了。

```
go test
PASS
    ok      _/home/go/src/golang-book-examples/hour15/example01      0.001s
```

如果对这个函数进行修改，使其检查 true 是否等于 false，测试将失败。

```
go test
--- FAIL:  TestTruth (0.00s)
    example01_test.go:7: The world is crumbling
FAIL
exit status 1
FAIL    _/home/go/src/golang-book-examples/hour15/example01      0.001s
```

测试失败时，命令 go test 提供了一些很有用的信息：测试名、文件名以及导致测试失败的代码所在行。这些信息有助于避免测试失败，因为它们明确地指出了问题出在什么地方。

TRY IT YOURSELF ▼

运行测试

在这个示例中，您将明白如何运行测试。

1. 在终端中切换到文件夹 hour15/example01。
2. 在终端中执行命令 go test。
3. 您将发现测试通过了。

▲

另一个约定是，在测试包中创建两个变量：got 和 want，它们分别表示要测试的值以及期望的值。程序清单 15.2 是一个简单的包，它返回一条问候语。

程序清单 15.2 一个返回问候语的简单包

```
 1:  package example02
 2:
 3:  import "testing"
 4:
 5:  func Greeting(s string) string {
 6:      return ("Hello " + s)
 7:  }
```

可对这个包进行测试，确认函数 Greeting 返回的字符串符合预期。程序清单 15.3 是一个针对函数 Greeting 的测试。

程序清单 15.3 got want 模式

```
 1:  package example01
 2:
 3:  import "testing"
 4:
 5:  func TestGreeting(t *testing.T) {
 6:      got := Greeting("George")
 7:      want := "Hello George"
 8:      if got != want {
 9:          t.Fatalf("Expected %q, got %q", want, got)
10:      }
11:  }
```

对程序清单 15.3 解读如下。

- 创建了一个名为 TestGreeting 的测试，它指出了测试针对的函数。这能够提高可读性，因为很轻易就能知道 TestGreeting 测试的是函数 Greeting。
- 将函数 Greeting 返回的值赋给了变量 got，这个变量表示要测试的值。
- 变量 want 表示期望的输出。
- 使用 if 语句检查 got 和 want 是否相同。如果不同，就引发错误。

如果您运行这个测试，将发现测试通过了。

```
go test
PASS
    ok      _/home/go/src/golang-book-examples/hour15/example02      0.001s
```

如果另一位开发人员在函数 Greeting 中将问候语改为 Hi，测试将以失败告终。这个示例很好地说明了如下两点：可指定期望输出；可通过测试确定情况发生了变化。

```
go test
--- FAIL:  TestGreeting (0.00s)
    example02_test.go:9: Expected "Hello George", got "Hi George"
FAIL
exit status 1
FAIL      _/home/go/src/golang-book-examples/hour15/example02      0.001s
```

got want 模式很有用，因为使用它可快速发现测试失败的原因。另外，通过在测试失败时显示有帮助的错误消息，可提高避免测试失败的速度。

▼ TRY IT YOURSELF

理解 got want 模式

在这个示例中，您将学会如何在 Go 语言测试中使用 got want 模式。

1. 在终端中切换到文件夹 hour15/example02。
2. 在终端中执行命令 go test。
3. 您将发现测试通过了。
4. 编辑文件 example02.go，将其中的 Hello 改为 Hi。
5. 在终端中再次执行命令 go test。
6. 注意到测试失败了，而终端中出现了相应的输出。

▲

15.3 运行表格驱动测试

通常，函数和方法的响应随收到的输入而异，在这种情况下，如果每个测试只使用一个值，将导致大量重复的代码。程序清单 15.4 扩展了程序清单 15.2，支持向函数 Greeting 传递区域，以显示相应语言的问候语。

程序清单 15.4 根据输入显示不同的输出

```
 1:  package example03
 2:
 3:  func translate(s string) string {
 4:      switch s {
 5:      case "en-US":
 6:          return "Hello "
 7:      case "fr-FR":
 8:          return "Bonjour "
 9:      case "it-IT":
```

```
10:          return "Ciao "
11:      default:
12:          return "Hello "
13:      }
14:  }
15:
16:  func Greeting(name, locale string) string {
17:      salutation := translate(locale)
18:      return (salutation + name)
19:  }
```

测试这个函数时，如果像程序清单 15.2 那样在每个测试中只使用一个值，则测试每个条件时都将包含大量重复的代码。

```
package example03

import "testing"

func TestFrTranslation(t *testing.T) {
    got := translate("fr")
    want := "Bonjour "
    if got != want {
        t.Fatalf("Expected %q, got %q", want, got)
    }
}

func TestUSTranslation(t *testing.T) {
    got := Greeting("George", "en-US")
    want := "Hello George"
    if got != want {
        t.Fatalf("Expected %q, got %q", want, got)
    }
}
```

对于这种情况，Go 语言提供了表格驱动测试模式，让您能够同时测试很多条件。程序清单 15.5 使用表格驱动测试模式重写了这个示例。

程序清单 15.5　使用表格驱动测试

```
 1:  package example04
 2:
 3:  import "testing"
 4:
 5:  type GreetingTest struct {
 6:      name   string
 7:      locale string
 8:      want   string
 9:  }
10:
11:  var greetingTests = []GreetingTest{
12:      {"George", "en-US", "Hello George"},
13:      {"Chloé", "fr-FR", "Bonjour Chloé"},
14:      {"Giuseppe", "it-IT", "Ciao Guiseppe"},
15:  }
16:
17:  func TestGreeting(t *testing.T) {
18:      for _, test := range greetingTests {
19:          got := Greeting(test.name, test.locale)
20:          if got != test.want {
21:          t.Errorf("Greeting(%s,%s) = %v; want %v", test.name, test.locale,
    actual, test.want)
22:      }
```

```
23: }
```

对程序清单 15.5 解读如下。

- 创建一个结构体，用于存储编写测试所需的数据，包括输入和期望的输出。
- 创建一个由结构体组成的切片，用于存储要测试的所有情形，包括期望输出。
- 在测试中，遍历切片中的所有结构体，并测试实际输出是否与期望输出相同。
- 只要有测试失败，就向控制台打印一条消息。

TRY IT YOURSELF

理解表格驱动测试

在这个示例中，您将学会如何使用表格驱动的测试。

1. 在终端中切换到文件夹 hour15/example04。
2. 在终端中执行命令 go test。
3. 您将看到测试通过了。

15.4 基准测试

第 9 章介绍了字符串以及如何拼接它们。您已经知道，拼接字符串的方法有多种，包括赋值、使用 join 附加以及使用缓冲区。第 9 章给出的建议是，使用缓冲区来拼接字符串，因为这种方法的性能最佳。这一点能够被证明吗？

Go 提供了功能强大的基准测试框架，能够让您使用基准测试程序来确定完成特定任务时性能最佳的方式是哪一种。程序清单 15.6 显示了 3 种拼接字符串的方式，请不要过度关注其中的函数，因为这里的重点是性能。

程序清单 15.6 3 种拼接字符串的方式

```
 1: package example04
 2:
 3: import (
 4:     "bytes"
 5:     "strings"
 6: )
 7:
 8: func StringFromAssignment(j int) string {
 9:     var s string
10:     for i := 0; i < j; i++ {
11:         s += "a"
12:     }
13:     return s
14: }
15:
16: func StringFromAppendJoin(j int) string {
17:     s := []string{}
```

```
18:      for i := 0; i < j; i++ {
19:          s = append(s, "a")
20:      }
21:      return strings.Join(s, "")
22:  }
23:
24:  func StringFromBuffer(j int) string {
25:      var buffer bytes.Buffer
26:      for i := 0; i < j; i++ {
27:          buffer.WriteString("a")
28:      }
29:      return buffer.String()
30:  }
```

这些函数根据传入的整数值生成相应长度的字符串。事实上，这些函数的功能完全相同。那么，如何确定哪种字符串拼接方式的性能是最佳的呢？

testing 包包含一个功能强大的基准测试框架，它能够让您反复地运行函数，从而建立基准。您无须指定运行函数的次数，因为基准测试框架将通过调整它来获得可靠的数据集。基准测试结束后，将生成一个报告，指出每次操作耗用了多少 ns。

基准测试名以关键字 Benchmark 打头，它们接受一个类型为 B 的参数，并对函数进行基准测试。程序清单 15.7 显示了分别对应于 3 种不同拼接方法的基准测试。

程序清单 15.7　高效地实现基准测试

```
 1:  package example05
 2:
 3:  import "testing"
 4:
 5:  func BenchmarkStringFromAssignment(b *testing.B) {
 6:      for n := 0; n < b.N; n++ {
 7:          StringFromAssignment(100)
 8:      }
 9:  }
10:
11:  func BenchmarkStringFromAppendJoin(b *testing.B) {
12:      for n := 0; n < b.N; n++ {
13:          StringFromAppendJoin(100)
14:      }
15:  }
16:
17:  func BenchmarkStringFromBuffer(b *testing.B) {
18:      for n := 0; n < b.N; n++ {
19:          StringFromBuffer(100)
20:      }
21:  }
```

这些基准测试使用循环反复地调用函数，以便建立基准。要运行这些测试，必须在命令 go test 中指定标志-bench。

```
go test -bench=.
BenchmarkStringFromAssignment-4          200000              5330 ns/op
BenchmarkStringFromAppendJoin-4          500000              2719 ns/op
BenchmarkStringFromBuffer-4             1000000              1213 ns/op
PASS
ok      _/home/go/src/golang-book-examples/hour15/example05     3.752s
```

这里运行了基准测试并显示基准值。从这些测试可知，赋值的性能最糟，使用 join 的

性能居中，而使用缓冲区的性能最佳。这个基准测试表明，使用缓冲区来拼接字符串的速度最快！

▼TRY IT YOURSELF

使用基准测试

在这个示例中，您将明白如何使用基准测试。

1. 在终端中切换到文件夹 hour15/example05。
2. 在终端中执行命令 go test --bench=.。
3. 您将看到基准测试的运行结果。

▲

15.5 提供测试覆盖率

测试覆盖率是度量代码测试详尽程度的指标，它指出了被测试执行了的代码所在的百分比值。回到前面的程序清单 15.2，假设在这个包中新增了一个函数，如程序清单 15.8 所示。

程序清单 15.8 扩展简单的包

```
 1:  package example02
 2:
 3:  import "testing"
 4:
 5:  func Greeting(s string) string {
 6:      return ("Hello " + s)
 7:
 8:  func Farewell(s string) string {
 9:      return ("Goodbye " + s)
10:  }
```

运行这个文件的测试时，结果表明测试通过了。

```
go test example06.go
PASS
ok      _/home/go/src/golang-book-examples/hour15/example06      0.002s
```

但这里存在一个问题：测试虽然通过了，但没有针对新函数 Farewell 的测试。在这种情况下，开发人员可能认为程序的运行情况符合预期，但实际情况并非如此。为避免这种情况发生，go test 命令提供了标志 -cover，可指出测试覆盖率。

```
go test-cover example06.go
PASS
coverage:  50.0% of statements
ok      _/home/go/src/golang-book-examples/hour15/example06      0.001s
```

上述输出表明，测试只覆盖了 50% 的代码。最佳目标是实现 100% 的覆盖，但实际上这样的目标并非总能实现，因为对于有些代码，要对其进行测试很难。建议您时不时地检查一

下测试覆盖率。

15.6 小结

本章介绍了测试。您学习了对代码进行测试的重要性，还有单元测试、功能测试和集成测试之间的差别；接下来，介绍了 TDD，一种先编写不能通过的测试，再编写代码的方法；然后，您学习了在 Go 语言中如何编写测试，还与 want got 测试模式；接下来，您学习了如何编写表格驱动测试，以支持测试多种输入；最后，您学习了如何对代码进行基准测试，以及如何检查测试覆盖率。阅读本章后，您明白了如何对 Go 代码进行测试，要成为卓有成效的 Go 语言程序员，这种能力至关重要。

15.7 问与答

问：真的需要编写测试吗？因为我编写的代码很少，而且没有时间。

答：从长远看，编写测试物有所值，即便项目很小。编写测试看似是负担，但实际上有很多好处。因为这样做有助于您理解代码、确保代码实现正确无误以及发现修改过程中引入的衰退。

问：该以什么样的频率运行测试？

答：理想情况下，每次提交代码前都应运行测试。有些开发人员在每次保存文件后都运行测试，但这可能比较麻烦，而且会分散注意力。提交代码前是运行测试的最佳时机。

问：应达到多高的测试覆盖率？

答：实现 100% 的测试覆盖率是一个值得为之努力的目标，但对大型项目而言，这几乎是不可能的。达到 80% 左右的测试覆盖率就可以了，具体多少取决于项目的复杂度。如果能够达到 100% 的覆盖率，就这样去做好了！

15.8 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

15.8.1 小测验

1. 应将测试文件放在什么地方？
2. 何为测试驱动开发（TDD）？
3. 基准测试有何好处？

15.8.2 答案

1. 测试文件与被测试文件放在相同的文件夹中，但在被测试文件名后加上了后缀_test。

2．测试驱动开发是这样一种做法，即先编写无法通过的测试来描述期望的功能，再着手编写代码。很多开发人员认为，这样做可提高代码设计质量，确保测试覆盖率成为开发过程的有机组成部分。

3．基准测试采用客观的经验方法来确定性能。使用基准测试意味着有可反复运行的测试，可用来确定一个函数的速度比另一个函数快，或者修改代码后性能提高或降低了。

15.9 练习

阅读 strings 包的源代码，查看其中的测试文件。您从中发现本章介绍的模式了吗？

第 16 章

调试

本章介绍如下内容。

- 使用日志帮助调试。
- 打印数据。
- 使用 **fmt** 包。
- 使用 **Delve**。
- 使用 **gdb**。

调试是确定程序为何不像预期那样工作的过程。程序不像预期那样工作的迹象有很多，包括编译错误、运行阶段错误、文件权限问题以及数据不正确等。调试是程序员经常需要做的工作，而要理解 Go 语言，就必须明白它提供了哪些工具。使用 Go 语言开发复杂的程序时，调试将成为日常工作中不可或缺的一部分。本章将介绍一些调试 Go 语言代码的方式。

16.1 日志

日志指的是记录程序执行期间发生的情况。无论程序需不需要调试，都会产生日志，这对于理解程序的执行情况很有帮助。很多常见的应用程序都提供了日志功能，这些日志可用来监视应用程序的健康状况、跟踪问题以及发现问题。

日志并非为报告 Bug 而提供的，而是可供在 Bug 发生时使用的基础设施。诸如 Nginx 等 Web 服务器在运行期间将日志写入文件，包括访问日志和错误日志。服务器管理员可使用这些日志来调试问题或核实是否一切正常。日志文件为文本文件，能够让您轻松地找到并理解特定的事件或错误。

每当用户请求服务器上的页面时，标准访问日志就会写入一行内容。

```
66.249.70.15 - - [27/Aug/2017:00:12:30 +0100] "GET /robots.txt HTTP/1.1" 200 12 "-"
"Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)"
```

这行内容提供了大量有关请求的信息。

- 远程 IP 地址。
- 日期和时间。
- HTTP 请求的类型。
- 请求的页面。
- 使用的 HTTP 协议。
- HTTP 响应代码。
- HTTP 客户端的用户代理。

这条日志表明，这是 Google 执行的一个正常操作，它向服务器请求文件 robots.txt，以确认能否对网站建立索引。在网站运行缓慢的情况下，这样的日志项很有用。如果访问日志表明在短时间内有大量来自同一个 IP 地址的请求，就意味着可能发生了意外的事情。在这种情况下，可采取措施，禁止相应的 IP 地址访问服务器。

Web 服务器 Nginx 还在它认为发生错误时将日志写入到一个独立的文件中。

```
2017/08/27 05:33:21 [error] 26706#26706: *322079 access forbidden by rule, client:
52.57.254.218, server:  shapeshed.com, request: "GET /wp-login.php HTTP/1.1", host:
"shapeshed.com"
```

在这个示例中，发生了被 Nginx 视为错误的事件。这条日志提供了有关错误的消息，它指出有人试图访问服务器上不允许访问的页面。这个客户端试图访问网站 WordPress（一个流行的开源博客引擎）的登录页面。很多恶意机器人都在服务器上扫描 WordPress 登录页面，然后试图访问服务器。这条日志表明，服务器上的规则禁止了请求，因此没有任何问题。

在 Web 服务器 Nginx 中，日志是一种未雨绸缪的调试方法，因为发生意外的事件时，有大量的信息可供用来调试问题。如果没有日志，问题将难以调试。另外，日志还可帮助完成应用程序的日常管理，确保程序像预期的那样运行。

Go 语言提供了 log 包，让应用程序能够将日志写入终端或文件。程序清单 16.1 是一个简单的程序，它输出一条日志消息。

程序清单 16.1　Go 语言提供的日志功能

```
 1:  package main
 2:
 3:  import "log"
 4:
 5:  func main() {
 6:      log.Printf("This is a log message")
 7:  }
```

这个程序将日志消息显示到终端。

```
go run example01.go
2017/08/27 07:17:46 This is a log message
```

注意到日志消息中包含日期和时间，这在您以后查看日志时很有用。log 包还可用来记录发生的致命错误。程序清单 16.2 生成了一个致命错误，并将其记录到了日志中。

程序清单 16.2 记录致命错误

```
 1:  package main
 2:
 3:  import (
 4:      "errors"
 5:      "log"
 6:  )
 7:
 8:  func main() {
 9:      var errFatal = errors.New("We only just started and we are crashing")
10:      log.Fatal(errFatal)
11:  }
```

虽然由于错误导致程序退出了，但是这个程序将一条日志消息写入终端。

```
go run example02.go
2017/08/27 07:26:13 We only just started and we are crashing
exit status 1
```

日志可用来了解程序的执行情况，它对调试能起到一定的作用，但是，查看日志对了解发生的意外事件更有帮助。这意味着需要将日志写入文件，以便以后能够访问它们。要将日志写入文件，可使用 Go 语言本身提供的功能，也可使用操作系统提供的功能。要将日志写入文件，只需命令 log 包这样做即可。

```
err := os.OpenFile("example03.log", os.O_APPEND|os.O_CREATE|os.O_RDWR, 0666)
if err != nil {
        log.Fatal(err)
}

defer f.Close()

log.SetOutput(f)
```

最后一行让 log 包使用文件 example03.log 来记录应用程序的日志。有关文件的处理，将在第 21 章更详细地介绍。程序清单 16.3 是一个完整的示例，演示了如何将日志消息写入文件。

程序清单 16.3 将日志写入文件

```
 1:  package main
 2:
 3:  import (
 4:      "log"
 5:      "os"
 6:  )
 7:
 8:  func main() {
 9:      f, err := os.OpenFile("example03.log", os.O_APPEND|os.O_CREATE|os.O_RDWR,
     0666)
10:      if err != nil {
11:          log.Fatal(err)
12:      }
13:
14:      defer f.Close()
15:
16:      log.SetOutput(f)
17:
18:      for i := 1; i <= 5; i++ {
19:          log.Printf("Log iteration %d", i)
20:      }
21:  }
```

▼ TRY IT YOURSELF

理解如何将日志写入到文件

在这个示例中，您将明白如何将日志写入文件。

1. 打开本书示例代码中的 hour16/example03.go。
2. 尝试理解这些代码是做什么的。
3. 在终端中执行命令 go run example03.go。
4. 在文件 example03.go 所在的文件夹中，您将看到一个名为 example03.log 的新文件。

▲

要将日志写入文件，还可使用操作系统提供的功能将日志输出从终端重定向到文件。这种方法不需要编写任何 Go 语言代码，因为它使用的是操作系统提供的功能。程序清单 16.4 所示的示例输出 5 条消息。

程序清单 16.4 日志程序示例

```
 1:  package main
 2:
 3:  import (
 4:      "log"
 5:  )
 6:
 7:  func main() {
 8:      for i := 1; i <= 5; i++ {
 9:          log.Printf("Log iteration %d", i)
10:      }
11:  }
```

以正常方式运行这个程序时，将向终端输出 5 条消息。然而，通过使用重定向功能，可将输出重定向到文件。在 Linux 和 Windows 系统中都可这样做。

```
go run example04.go > example04.log 2>&1
```

通常，最好使用操作系统将日志重定向到文件，而不要使用 Go 语言代码。因为这种方法更灵活，能够让其他工具在必要时使用日志。

16.2 打印数据

在很多情况下，Bug 都是由于数据不符合预期导致的。程序清单 16.5 是一个简单的程序，它提示玩家输入，并根据输入判断玩家能否获得奖励。如果玩家猜对了名字，就将获得奖励；否则将显示一条消息，指出他没有获奖。

程序清单 16.5 简单的猜名游戏

```
 1:  package main
 2:
 3:  import (
```

```
    "bufio"
    "fmt"
    "os"
    "strings"
)

func main() {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print("Guess the name of my pet to win a prize:  ")
    text, _ := reader.ReadString('\n')
    text = strings.Replace(text, "\n", "", -1)

    if text == "John" {
        fmt.Println("You won! You win chocolate!")
    } else {
        fmt.Println("You didn't win. Better luck next time")
    }
}
```

作为这个程序的开发者，您得知有人知道您的宠物的名字，却没有获奖。这个人指出他正确地输入了宠物的名字，您询问他输入的名字是否是 John，而对方肯定了这一点。现在该调试代码了！就这里而言，要确定为何会这样，最快捷、最简单的方式是将数据打印出来。为调试这个程序，一种非常简单的方式是，在 if 语句前面添加一行将输入的数据打印出来的代码。

```
fmt.Println("[DEBUG] text is:", text)
```

现在如果运行这个程序，它将打印输入的数据。

```
go run example06.go
Guess the name of my pet to win a prize:  john
[DEBUG] text is:  john
You didn't win. Better luck next time
```

通过打印变量来调试代码，您发现玩家输入的是 john 而不是 John。由于 if 语句要求名字的首字母大写，因此这条 if 语句的结果为 false，玩家没能获得奖励。通过使用这种技巧，您很快就发现了 Bug。要修复这个 Bug，可在这个程序中添加代码，将用户输入转换为小写，再将其与 john 进行比较。这意味着不再区分大小写。

像这样打印数据的做法通常被视为一种快速解决方案，因为只需添加打印数据的代码行，就可能快速发现 Bug。然而，使用这种方法将导致代码中充斥调试代码。

16.3 使用 fmt 包

为将值打印到终端，有必要探索一下 fmt 包以及如何将其用于调试。fmt 包可用来设置格式，因此必要时可使用它来输出数据，以方便调试。通过使用函数 Printf，可创建要打印的字符串，并使用百分符号在其中引用变量。fmt 包将对变量进行分析，并输出字符串。程序清单 16.6 是一个简单的示例，它使用函数 Printf 将调试语句输出到终端。

程序清单 16.6 使用 fmt 包来调试代码

```
package main

import (
```

```
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      s := "Hello World"
 9:      fmt.Printf("String is %v\n", s)
10:  }
```

▼TRY IT YOURSELF

使用 fmt 包来调试代码

在这个示例中，您将明白如何使用 fmt 包来调试代码。

1. 打开本书代码示例中的 hour16/example07.go。
2. 尝试理解这些代码是做什么的。
3. 在终端中执行命令 go run example07.go。
4. 您将在终端中看到如下调试语句。

```
String is Hello World
```

▲

请注意，这个字符串末尾是字符\n，这表示换行，因为函数 Printf 默认不会添加回车。动词%v 是类型的默认格式。

也可在同一行中引用多个变量。程序清单 16.7 使用了两个变量来生成调试语句。将按变量出现的顺序对其进行分析。

程序清单 16.7 使用 fmt 包打印多个变量

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      s := "Hello World"
 9:      t := "Goodbye, Cruel World"
10:      fmt.Printf("s is %v, t is %v\n", s, t)
11:  }
```

▼TRY IT YOURSELF

使用 fmt 包打印多个变量

在这个示例中，您将明白如何使用 fmt 包来打印多个变量。

1. 打开本书代码示例中的 hour16/example08.go。
2. 尝试理解这些代码是做什么的。

3. 在终端中执行命令 go run example08.go。
4. 您将在终端中看到如下调试语句。

```
s is Hello World, t is Goodbye, Cruel World
```

▲

第 8 章说过，可结合使用动词 v 和 + 来打印结构体中字段的名称。这很有用，因为这样您无须查找字段名，也无须记住结构体中字段的排列顺序。

```
fmt.Printf("%+v\n", someStruct)
```

程序清单 16.8 演示了如何将结构体打印到控制台，包括打印字段名和不打印字段名。

程序清单 16.8　使用 fmt 包打印结构体的值

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  type Animal struct {
 8:      Name string
 9:      Color string
10:  }
11:
12:  func main() {
13:      a := Animal{
14:          Name:  "Cat",
15:          Color:  "Black",
16:      }
17:      fmt.Printf("%v\n", a)
18:      fmt.Printf("%+v\n", a)
19:  }
```

TRY IT YOURSELF ▼

使用 fmt 包打印结构体的值

在这个示例中，您将明白如何打印结构体的值。

1. 打开本书代码示例中的 hour16/example09.go。
2. 尝试理解这些代码是做什么的。
3. 在终端中执行命令 go run example09.go。
4. 您将在终端中看到如下调试语句。

```
{Cat Black}
{Name:Cat Color:Black}
```

▲

16.4 使用 Delve

Go 语言没有官方调试器，但很多社区项目都提供了 Go 语言调试器。Delve 就是一个这样的项目，它为 Go 项目提供了丰富的调试环境。要安装 Delve，可像下面这样做。

```
go get github.com/derekparker/delve/cmd/dlv
```

要检查是否正确地安装了 Delve，可在终端中执行命令 dlv --help，您能在终端中看到一些帮助文本。相比于日志和打印代码行，Delve 提供的环境要精致得多。虽然使用 fmt 包进行调试是一种快捷而简单的方法，但 Delve 提供了更精致的程序调试环境。Delve 让开发人员能够与正在执行的程序交互、进入正在运行的进程以及查看核心转储和栈跟踪。程序清单 16.9 是一个简单的程序，我们将使用它来演示如何使用 Delve。

程序清单 16.9 用来演示如何使用 Delve 的简单程序

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:  )
 6:
 7:  func main() {
 8:      s := "Hello World"
 9:      t := "Goodbye Cruel World"
10:      echo(s)
11:      echo(t)
12:  }
```

可使用 Delve 来执行这个程序，并暂停执行以便进行调试。

```
dlv debug example10.go
Type 'help' for list of commands.
(dlv)
```

实际上，这个控制台位于程序中，且已暂停执行：等待您恢复执行。假设我们对传递给函数 echo 的变量 s 感兴趣，可使用工具 Delve 让程序每次调用函数 echo 时都暂停执行。这被称为断点，意味着在指定的地方暂停执行程序。这样做很有用，因为您可查看并修改变量，再让程序接着执行。要让 Delve 在函数调用处设置断点，可像下面这样做。

```
(dlv) break echo
Breakpoint 1 set at 0x47b7b8 for main.echo() ./example10.go:7
```

这样每次调用函数 echo 前，程序都将暂停执行，让调试器能够查看程序的状态。可在程序中添加任意数量的断点。创建所有的断点后，就可使用 continue 来执行程序了。

```
(dlv) continue
> main.echo() ./example10.go:7 (hits goroutine(1):1 total:1) (PC:  0x47b7b8)
2:
3:  import (
4:      "fmt"
5:  )
```

```
 6:
 =>   7:  func echo(s string) {
 8:       fmt.Println(s)
 9:       return
10:  }
11:
12:  func main() {
```

执行到第一个断点时，程序暂停执行。工具 Delve 打印了断点的上下文，这可以让您知道函数 echo 被调用。还可查看此时的程序状态——使用 print 将变量 s 的值打印出来。

```
(dlv) print s
"Hello World"
```

由于这是函数 echo 首次被调用，因此变量 s 的值为 Hello World，这符合预期。要接着往下执行程序，可再次使用命令 continue。

```
 (dlv) continue
 Hello World
 > main.echo() ./example10.go:7 (hits goroutine(1):2 total:2) (PC:  0x47b7b8)
 2:
 3:  import (
 4:          "fmt"
 5:  )
 6:
=>    7: func echo(s string) {
 8:          fmt.Println(s)
 9:          return
10:  }
11:
12:  func main() {
```

注意到这里打印了首次调用函数 echo 得到的输出，因为程序的这部分代码已经执行。接下来，再次调用函数 echo 时程序又暂停执行。这次打印变量 s 的值时，显示的结果为 Goodbye Cruel World。

```
(dlv) print s
"Goodbye Cruel World"
```

再次执行命令 continue 让程序接着执行。由于没有更多的断点，程序退出。

```
(dlv) continue
Goodbye Cruel World
Process 7094 has exited with status 0}
```

TRY IT YOURSELF ▼

使用 Delve 来调试代码

在这个示例中，您将明白如何使用 Delve 来调试 Go 语言代码。

1. 使用下面的命令安装 Delve。

```
go get github.com/derekparker/delve/cmd/dlv
```

2. 开始调试本书代码示例中 hour16/example10.go 的代码。

```
dlv debug example10.go
```

3. 逐步执行这个示例：设置断点并打印变量的值。
4. 接着往下执行，直到程序退出。

16.5 使用 gdb

如果您使用的是 UNIX 型系统（macOS 或 Linux），可使用 GNU 调试器来调试 Go 程序。GNU 调试器应用广泛，因此通常可使用包管理器来安装。GNU 调试器操作的是二进制文件，因此需要先编译 Go 程序以生成二进制文件。

```
go build example10.go
```

这将创建一个名为 example10 的二进制文件。要使用 GNU 调试器进行调试，首先需要像下面这样启动它。

```
gdb example10
```

您将看到大量的输出，最后进入显示程序已暂停执行的控制台。使用 GNU 调试器时，可使用命令 list 来查看代码的组成部分。

```
list main.echo
(gdb) l main.echo
2
3       import (
4               "fmt"
5       )
6
7       func echo(s string) {
8               fmt.Println(s)
9               return
10       }
11
```

这里显示了函数 echo 的上下文。要设置断点，可使用命令 break。执行这个命令时，可指定行号，也可指定函数名。

```
break main.echo
```

要开始执行程序，可使用命令 run。这个命令接着往下执行程序，直到遇到第一个断点。此时与使用 Delve 工具时一样，可查看变量 s 的值。

```
(gdb) print s
$1 = 0x4a5891 "Hello World"
```

此时可使用命令 continue 接着往下执行程序。这时将执行到第二个断点处——再次调用函数 echo 代码。如果此时查看变量 s 的值，将发现它发生了变化，这符合预期。

```
(gdb) print s
```

```
Goodbye Cruel World
$2 = 0x4a6831 "Goodbye Cruel World"
```

GNU 调试器提供了丰富的调试环境，但它并非专用于 Go 语言。它提供了极细的调试粒度，让您能够逐行地执行代码。

16.6 小结

本章介绍了很多调试 Go 程序的方法。首先介绍了 log 包以及日志为何是一种很有用的未雨绸缪的调试方法。接下来，您探索了 fmt 包，得知使用它可将变量的值输出到控制台，而这是一种实用的完成简单调试任务的方法。然后，介绍了两种第三方工具，它们提供了更丰富、交互性更强的调试环境。首先，您得知社区项目 Delve 支持暂停执行程序以及设置断点；其次，您学习了 GNU 调试器，这是一个成熟的开源工具，它提供了与 Delve 类似的工具——暂停执行程序以及逐步执行程序。本章介绍了一些很有用的调试工具，但学习调试的最佳途径是真刀真枪地去调试 Bug！有了本章介绍的工具，您应该能够着手去调试 Bug 了。

16.7 问与答

问：我适合使用哪种调试工具？

答：要进行多详尽的调试，取决于程序的复杂程度以及您多想学习诸如 Delve 和 gdb 等工具的用法。就快速调试而言，通常使用 fmt 包将变量的值打印出来就能修复 Bug。然而，随着程序越来越复杂，您可能需要使用功能更齐备的调试器。

问：使用 fmt 或 log 包是否会导致代码中充斥调试语句？

答：确实，这会导致代码中充斥调试语句。这些语句可能很有用，但您也可选择在完成调试后将它们删除。有些第三方的日志工具支持不同的模式，让应用程序能够在调试模式下运行以增大日志量。如果您真的不希望程序中包含调试代码，可考虑使用诸如 Delve 和 gdb 等调试器。

问：测试在调试中扮演着什么角色？

答：第 15 章介绍了如何编写测试。发现 Bug 后，最好编写不能通过的相应测试。这样在调试代码的过程中，可再次运行这个测试，看看它能否通过。这样做还有另一个好处，那就是以后修改代码时如果引入了 Bug，您将有相应的测试。

16.8 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

16.8.1 小测验

1. 为何日志对调试很有帮助？

2．什么是断点？

3．相比于 Delve 等工具，使用日志有何优缺点？

16.8.2 答案

1．在程序中发现 Bug 时，日志可提供很有用的信息。如果程序在整个执行期间都记录日志，将对调试大有帮助，因为发现 Bug 时，可通过日志文件获悉程序的状态。在少数情况下，日志可直接提供答案。

2．断点让程序执行到相应的位置时暂停。根据使用的调试器，您可能能够查看整个程序的状态，包括变量的值、调用栈和内存分配情况。断点只导致程序暂停，这意味着可接着往下执行程序。

3．诸如 Delve 等工具提供了丰富的调试环境，但要使用它们，则需要花点时间学习。这些工具能够暴露 Go 语言的内部情况以及它是如何与操作系统交互的，因此可能有一些新的东西需要学习。从长远看，这样做是值得的，但就简单调试而言，使用日志或将值打印到终端可能就足够了。

16.9 练习

使用 Delve 或 gdb 与您编写的一个 Go 程序交互。在特定函数或行号处设置断点，并逐步执行程序。使用命令 help 来了解调试器提供的一些功能。

第 17 章

使用命令行程序

本章介绍如下内容。

- 操作输入和输出。
- 访问命令行参数。
- 分析命令行标志。
- 自定义帮助文本。
- **POSIX** 兼容性。
- 安装和分享命令行程序。

命令行程序也叫命令行实用程序或工具，它被设计在终端运行。本章介绍如何使用 Go 语言创建命令行程序，您将了解如何分析参数以及如何创建子命令。

在图形用户界面（GUI）面世前，与计算机交互通常是通过命令行进行的。当前，对程序员和系统管理员来说，命令行程序依然是一种流行而实用的与底层操作系统交互的方式。出于如下原因，程序员想创建命令行程序。

- 为创建能够定期自动运行的脚本。
- 为创建与系统中的文件交互的脚本。
- 为创建能够执行系统维护任务的脚本。
- 为避免设计图形用户界面这种无谓的开销。

命令行程序通常执行下面这样的操作：管理目录中的文件或接受一些输入数据并返回一些输出数据。一个这样的典型示例是 Windows、Linux 和 macOS 都支持的命令 sort，这个命令接受一个每行都包含单词的文件，并返回排序后的版本。假设文件 beatles.txt 包含如下文本。

```
John
Paul
Ringo
George
```

通过将这个文件作为输入数据提供给命令 sort，将返回这个文件的排序版本，并将其打印到终端。

```
$ sort beatles.txt
George
John
Paul
Ringo
```

命令行程序可使用任何编程语言来编写，只要脚本是可执行的，就可使用终端来运行（执行）它。命令行程序还可由操作系统自动运行，由操作系统自动运行的脚本可能这样做。

- 每隔 1min 从 Web 服务那里取回数据。
- 每隔 1h 删除临时文件。
- 每天都备份数据库。
- 每个月都执行系统维护任务。
- 每年都提醒您的生日。

17.1 操作输入和输出

编写命令行程序前，必须明白一些理论，确保您编写的脚本能够与操作系统和其他脚本交互。命令行程序操作输入和输出。对于这些输入和输出，Windows、macOS 和 Linux 使用的术语相同，而 Go 语言也使用这些术语。鉴于此，熟悉有关输入和输出的术语和代码很有用，表 17.1 总结了这些代码及其含义。

表 17.1 代码及其含义

名称	代码	描述
标准输入	0	包含提供给程序的输入
标准输出	1	包含显示到屏幕上的输出
标准错误	2	包含显示到屏幕上的错误消息

标准输入是提供给命令行程序的数据，它可以是文件，也可以是文本字符串。在前面的 sort 示例中，文件 beatles.txt 就是标准输入。

标准输出是来自程序的输出，在前面的 sort 示例中，经过排序的数据被打印到终端。标准错误是来自程序的错误，前面的 sort 示例没有错误，但如果文件 beatles.txt 不存在，将向标准错误打印一条错误消息，以指出这一点。

长期运行的进程（如 Web 服务器）常常将数据同时记录到标准输入和标准输出，即通常将数据发送到日志文件。与命令行程序相关的生态系统几乎都是从正确地使用标准流（即标准输入、标准输出和标准错误）的程序衍生而来的，因此对标准流有大致的了解将对您大有帮助。

17.2 访问命令行参数

在创建命令行程序方面，Go 语言提供了强大的支持。它遵循接受输入并发送输出的理念，

且通常会自动确保输出被发送到正确的输出流。在命令行中传递给命令行程序的数据被称为参数。在 Go 语言中，要读取传递给命令行程序的参数，可使用标准库中的 os 包，如程序清单 17.1 所示。

程序清单 17.1 访问命令行参数

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "os"
 6:  )
 7:
 8:  func main() {
 9:      for i, arg := range os.Args {
10:          fmt.Println("argument", i, "is", arg)
11:      }
12:  }
```

方法 Args 返回一个字符串切片，其中包含程序的名称以及传递给程序的所有参数。在这个示例中，使用了 range 来遍历参数并将其打印到终端。在本章中，不使用命令 go run 来执行程序，而先使用 go build 来构建程序，再将其作为可执行文件来运行。这旨在让您熟悉命令行程序是可执行文件的概念，同时突出不同平台之间的一些细微差别。

将前述示例保存为文件 example01.go 后，就可构建并运行它。在 Windows 系统中，运行命令 go build 将生成一个扩展名为.exe 的文件，而在 Linux 和 macOS 系统中，执行这个命令将生成一个没有扩展名的可执行文件。请注意，在 Windows 和 Linux/macOS 系统中，执行文件的方式存在细微的差别。在 Linux/macOS 系统中，要执行当前工作目录下的可执行文件，必须在指定文件时加上前缀./；而在 Windows 系统中，则无须添加这样的前缀。

```
# Linux / macOS
$ go build example01.go
$ ./example01
argument 0 is ./example01

# Windows
$ go build example01.go
$ example01
argument 0 is example01
```

TRY IT YOURSELF ▼

访问参数

在这个示例中，您将明白如何使用参数。

1. 在文本编辑器中打开文件 hour17/example01.go，尝试理解这个示例是做什么的。
2. 使用命令 go build example01.go 构建这个程序。
3. 运行这个示例。

```
# Linux
./example01
# Windows
```

```
example01
```

4. 您将看到第一个参数为可执行文件的名称。

17.3 分析命令行标志

虽然可使用 os 包来获取命令行参数，但 Go 语言还在标准库中提供了 flag 包。除 os.Args 的功能外，这个包还提供了众多其他的功能，其中包括以下几点。

- 指定作为参数传递的值的类型。
- 设置标志的默认值。
- 自动生成帮助文本。

程序清单 17.2 所示的简单程序演示了 flag 包的用法。

程序清单 17.2 分析命令行标志

```
 1:  package main
 2:
 3:  import (
 4:      "flag"
 5:      "fmt"
 6:  )
 7:
 8:  func main() {
 9:      s := flag.String("s", "Hello world", "String help text")
10:      flag.Parse()
11:      fmt.Println("value of s:", *s)
12:  }
```

对这个程序解读如下。

- 声明变量 s 并将其设置为 flag.String 返回的值。
- flag.String 能够让您声明命令行标志，并指定其名称、默认值和帮助文本。
- 调用 flag.Parse，让程序能够传递声明的参数。
- 最后，打印变量 s 的值。请注意，flag.String 返回的是一个指针，因此使用运算符* 对其解除引用，以便显示底层的值。

如果您构建并运行这个程序，将发现 flag 包给标志 -s 设置了默认值。

```
$ go build example02.go
$ ./example02
value of s:  Hello World
```

运行这个程序时，也可给标志 -s 指定值。

```
$ ./example02 -s Goodbye
value of s:  Goodbye
```

flag 包会自动创建一些帮助文本，要显示它们，可使用如下任何标志。

- -h。
- --h。
- -help。
- --help。

帮助文本被格式化，并向最终用户指出了应将参数设置为什么样的值。

```
$ ./example02 -h
Usage of ./example02:
 -s string
   String help text (default "Hello world")
```

在本章余下的篇幅中，将只演示在 Linux/macOS 系统中是如何做的。如果您使用的是 Windows 系统，则只需在运行可执行文件时省略文件名前面的./即可。

运行可执行文件 example01 时，可传递多个参数。为此，可先指定这个可执行文件的名称，再指定用空格分隔的参数。程序将收到这些参数，并将它们打印到终端。

```
$ go build example01.go
$ ./example01 foo bar baz
argument 0 is ./example01
argument 1 is foo
argument 2 is bar
argument 3 is baz
```

17.4 指定标志的类型

flag 包根据声明分析标志的类型，这对应于 Go 语言的类型系统。编写命令行程序时，必须考虑程序将接受的数据，并将其映射到正确的类型，这一点很重要。程序清单 17.3 演示了如何分析 String、Int 和 Boolean 标志，并将它们的值打印到终端。

程序清单 17.3 分析 String、Int 和 Boolean 标志

```
 1:  package main
 2:
 3:  import (
 4:      "flag"
 5:      "fmt"
 6:  )
 7:
 8:  func main() {
 9:      s := flag.String("s", "Hello world", "String help text")
10:      i := flag.Int("i", 1, "Int help text")
11:      b := flag.Bool("b", false, "Bool help text")
12:      flag.Parse()
13:      fmt.Println("value of s:", *s)
14:      fmt.Println("value of i:", *i)
15:      fmt.Println("value of b:", *b)
16:  }
```

如果您构建并运行这个示例，将发现正确地设置了这些标志的值。

```
$ go build example03.go
$ ./example03
value of s:  Hello world
```

```
value of i:  1
value of b:  false
```

执行这个示例时，可通过传入值来修改标志的值。请注意，对于 Boolean 标志，如果仅指定它，将把它的值设置为 true。

```
$ ./example03 -s Goodbye -i 42 -b
value of s:  Goodbye
value of i:  42
value of b:  true
```

如果用户执行命令行程序时给标志指定的值的类型不正确，将显示错误消息。在下面的示例中，将标志-i 设置成了字符串 String。由于必须将这个标志设置为整数值，因此将导致错误：flag 包的默认行为是打印错误消息并显示帮助文本。

```
$ ./example03 -i String
invalid value "String" for flag -i:  strconv.ParseInt:  parsing "String":  invalid
syntax
  Usage of ./example03:
    -b Bool help text
    -i int
      Int help text (default 1)
    -s string
      String help text (default "Hello world")
```

17.5 自定义帮助文本

虽然 flag 包会自动生成帮助文本，但完全可以覆盖默认的帮助格式并提供自定义的帮助文本。为此可将变量 Usage 设置为一个函数，这样每当在分析标志的过程中发生错误时，都将调用这个函数。下面是这个函数的一种简单实现。

```
flag.Usage = func() {
    fmt.Fprintln(os.Stderr, "hello world")
}
```

请注意，这里使用了标准库中的 os 包来将消息打印到标准误差（standard Error），因为这条消息将在发现分析错误时显示，但输出是完全可定制的。

在前面的示例中，可将变量 Usage 指定为一个自定义函数，如程序清单 17.4 所示。

程序清单 17.4 给命令行工具创建帮助文本

```
 1:  package main
 2:
 3:  import (
 4:      "flag"
 5:      "fmt"
 6:      "os"
 7:  )
 8:
 9:  func main() {
10:      flag.Usage = func() {
11:          usageText := `Usage example04 [OPTION]
12:  An example of customizing usage output
13:
14:  -s, --s          example string argument, default:  String help text
15:  -i, --i          example integer argument, default:  Int help text
```

```
16:  -b, --b          example boolean argument, default:  Bool help text`
17:        fmt.Fprintf(os.Stderr, "%s\n", usageText)
18:      }
19:
20:      s := flag.String("s", "Hello world", "String help text")
21:      i := flag.Int("i", 1, "Int help text")
22:      b := flag.Bool("b", true, "Bool help text")
23:      flag.Parse()
24:      fmt.Println("value of s:", *s)
25:      fmt.Println("value of i:", *i)
26:      fmt.Println("value of b:", *b)
27:  }
```

通过使用原始字符串字面量（放在`和`之间的内容），将保留字符串的格式设置。现在如果您构建并运行这个程序，将显示自定义的帮助文本。

```
$ go build example04.go
$ ./example04 --help
Usage example04 [OPTION]
An example of customizing usage output

  -s, --s         example string argument, default:  String help text
  -i, --i         example integer argument, default:  Int help text
  -b, --b         example boolean argument, default:  Bool help text
```

TRY IT YOURSELF ▼

在命令行程序中设置帮助文本

在这个示例中，您将自定义帮助文本。

1. 在文本编辑器中打开文件 hour17/example04.go，尝试理解这个示例是做什么的。
2. 使用命令 go build example04.go 构建这个程序。
3. 运行这个示例并指定选项 -h。

```
# Linux
./example04 -h
# Windows
example04 -h
```

4. 注意到显示了自定义的帮助文本。

▲

17.6 创建子命令

很多命令行程序都支持子命令，一个典型的示例是 git，它包含顶级命令 git 和多个子命令，而这些子命令都有独立的选项和帮助文本。下面是 git 的一些子命令。

```
git clone
git branch
```

如果运行这些子命令时指定标志 --help，您将发现它们有独立的选项。flag 包通过 FlagSets

提供了子命令支持，让您能够创建子命令，并指定独立的标志集。要创建子命令并指定标志，可像下面这样做。

```
cloneCmd := flag.NewFlagSet("clone", flag.ExitOnError)
```

其中第一个参数为子命令名，而第二个参数则指定了错误处理行为。

- flag.ContinueOnError：如果没有分析错误，就继续执行。
- flag.ExitOnError：如果有分析错误，就退出并将状态码设置为 2。
- flag.PanicOnError：如果发生分析错误，就引发 panic。

使用 NewFlagSet 可创建独立的标志集。要根据参数做相应的处理，可使用 switch 语句。本章开头指出过，os.Args 包含原始参数，因此可在 switch 语句中使用它来处理标志集。请注意，由于索引从 0 开始，因此这里需要使用索引 1。

```
switch os.Args[1] {
    case "clone":
    // 这里处理 clone 子命令
    case "branch":
    // 这里处理 branch 子命令
    default:
    // 这里处理其他条件
}
```

在下面的示例中，创建了一个命令行工具，它提供了两个命令：uppercase 和 lowercase。这些命令接受标志 -s 或 --s 指定的字符串，并返回处理后的文本。当然，这个示例很简单，但这里的重点是演示如何创建子命令，而不是复杂的逻辑。

```
uppercaseCmd := flag.NewFlagSet("uppercase", flag.ExitOnError)
lowercaseCmd := flag.NewFlagSet("lowercase", flag.ExitOnError)

switch os.Args[1] {
    case "uppercase":
        s := uppercaseCmd.String("s", "", "A string of text to be uppercased")
        uppercaseCmd.Parse(os.Args[2:])
        fmt.Println(strings.ToUpper(*s))
    case "lowercase":
        s := lowercaseCmd.String("s", "", "A string of text to be lowercased")
        lowercaseCmd.Parse(os.Args[2:])
        fmt.Println(strings.ToLower(*s))
    default:
        // 这里处理其他条件
}
```

对这个代码片段解读如下。

- 创建了两个 FlagSet，一个表示命令 uppercase，另一个表示命令 lowercase。
- 使用 switch 语句读取命令的第一个参数。
- 如果这个参数为 uppercase，就在 FlagSet uppercase 中初始化一个字符串标志，再将其他参数传递给 FlagSet uppercase，并对它们进行分析。
- 将 s 的值传递给 strings 包中的方法 ToUpper，以便将其转换为大写。如果用户没有给 s 指定值，将传递默认的空字符串。

- 对于 FlagSet lowercase，逻辑与此相同。
- 如果第一个参数既不是 uppercase，也不是 lowercase，将执行 switch 语句的 default 部分，但这里没有做任何处理。

当前，在这个程序的顶层什么都没有做。例如，如果用户执行这个程序时没有提供任何参数，则什么都不会做。请看看 git 命令在用户没有提供任何子命令时是如何做的，它显示有关各个子命令的帮助信息。为实现这样的行为，可设置变量 Usage，并在 os.Args 的长度为 1 时调用它，如程序清单 17.5 所示。因为 os.Args 的长度为 1 时，说明用户只指定了可执行文件名。

程序清单 17.5　引入子命令

```
 1:  package main
 2:
 3:  import (
 4:      "flag"
 5:      "fmt"
 6:      "os"
 7:      "strings"
 8:  )
 9:  func flagUsage() {
10:      usageText := `example05 is an example cli tool.
11:
12:  Usage:
13:  example05 command [arguments]
14:  The commands are:
15:  uppercase uppercase a string
16:  lowercase lowercase a string
17:  Use "example05 [command] --help" for more information about a command.`
18:      fmt.Fprintf(os.Stderr, "%s\n\n", usageText)
19:  }
20:
21:  func main() {
22:
23:      flag.Usage = flagUsage
24:      uppercaseCmd := flag.NewFlagSet("uppercase", flag.ExitOnError)
25:      lowercaseCmd := flag.NewFlagSet("lowercase", flag.ExitOnError)
26:
27:      if len(os.Args) == 1 {
28:          flag.Usage()
29:          return
30:      }
31:  }
```

现在如果用户只输入了命令，而没有指定任何参数，则用户将看到一些有关如何使用该命令的帮助信息。完整的示例见程序清单 17.6。

程序清单 17.6　在命令行程序中使用子命令

```
 1:  package main
 2:
 3:  import (
 4:      "flag"
 5:      "fmt"
 6:      "os"
 7:      "strings"
 8:  )
 9:  func flagUsage() {
10:      usageText := `example05 is an example cli tool.
```

```
11:
12:  Usage:
13:  example05 command [arguments]
14:  The commands are:
15:  uppercase uppercase a string
16:  lowercase lowercase a string
17:  Use "example05 [command] --help" for more information about a command.`
18:      fmt.Fprintf(os.Stderr, "%s\n\n", usageText)
19:  }
20:
21:  func main() {
22:
23:      flag.Usage = flagUsage
24:      uppercaseCmd := flag.NewFlagSet("uppercase", flag.ExitOnError)
25:      lowercaseCmd := flag.NewFlagSet("lowercase", flag.ExitOnError)
26:
27:      if len(os.Args) == 1 {
28:          flag.Usage()
29:          return
30:      }
31:
32:      switch os.Args[1] {
33:          case "uppercase":
34:              s := uppercaseCmd.String("s", "", "A string of text to be
    uppercased")
35:              uppercaseCmd.Parse(os.Args[2:])
36:              fmt.Println(strings.ToUpper(*s))
37:          case "lowercase":
38:              s := lowercaseCmd.String("s", "", "A string of text to be
    lowercased")
39:              lowercaseCmd.Parse(os.Args[2:])
40:              fmt.Println(strings.ToLower(*s))
41           default:
42:              flag.Usage()
43:      }
44:  }
```

如果您构建并执行这个程序，将发现每个子命令都将返回正确的文本。

```
$ ./example05 uppercase -s "i want to grow up"
I WANT TO GROW UP
$ ./example05 lowercase -s "I DO NOT WANT TO GROW UP"
i do not want to grow up
```

17.7 POSIX 兼容性

在 Linux 和 macOS 系统中，大多数命令行工具都要求以推荐标准 POSIX 指定的方式传递参数。POSIX 是一系列标准，旨在确保操作系统之间彼此兼容。很多开发人员都希望采用这种方式，因此关于 Go 语言的 flag 包的话题中，经常会引发对 POSIX 的讨论。

虽然 Go 语言的 flag 包没有遵循这些推荐标准，但有多个第三方替代品遵循了这些推荐标准。

17.8 安装和分享命令行程序

开发好命令行程序后，请在您的系统中安装它，以便能够在任何地方，而不是只能在命令 go build 生成的二进制文件所在的文件夹中才能访问它。要让 Go 工具发挥作用，必须遵

循 Go 语言约定，这很重要。为此，必须正确地设置$GOPATH。使用标准的目录布局，并将代码放在一个位于 src 的子文件夹中。

```
.
├── bin
├── pkg
└── src
    └── github.com
```

如果您采取了第 1 章推荐的做法，文件夹 github.com 将有一个以您的 Github 用户名命名的子文件夹。对我来说，这个子文件夹的路径如下所示，但您需要将 shapeshed 改为您的 Github 用户名。

```
$GOPATH/src/github.com/shapeshed/
```

您的 Go 项目位于这个文件夹中。请在这个文件夹中创建一个名为 helloworld 的文件夹。

```
// Linux / macOS
$ mkdir -p $GOPATH/src/github.com/[your github username]/helloworld
// Windows
$ mkdir "%GOPATH%\src\github.com\[your github username]/helloworld"
```

切换到这个文件夹，并在其中创建一个名为 helloworld.go 的文件，再在这个文件中添加如程序清单 17.7 所示的代码。

程序清单 17.7　访问命令行参数

```
1:  package main
2:
3:  import "fmt"
4:
5:  func main() {
6:      fmt.Println("Hurray! You are a Gopher!")
7:  }
```

现在可以使用命令 go install 安装这个程序了。请注意，指定的路径必须是相对于 $GOPATH 的。

```
go install github.com/[your github username]/helloworld
```

如果一切顺利，则现在应该能够在系统的任何地方执行 helloworld 了。

```
helloworld
Hurray! You are a Gopher!
```

遵循 Go 语言的约定在于，您现在可以将代码提交到 Github，让别人能够使用下面的命令轻松地安装它。

```
go get github.com/[your github username]/helloworld
```

TRY IT YOURSELF ▼

安装命令行程序

在这个示例中，您将安装一个命令行程序。

1. 在文本编辑器中打开文件 hour17/example07.go，并尝试理解这个示例是做什么的。
2. 在$GOPATH 下创建一个以您的 Github 用户名命名的文件夹。

```
// Linux / macOS
mkdir -p $GOPATH/src/github.com/[your github username]/helloworld
// Windows
mkdir "%GOPATH%\src\github.com\[your github username]/helloworld"
```

3. 将文件 example07.go 复制到这个文件夹。
4. 安装这个程序。

```
go install github.com/[your github username]/helloworld
```

5. 运行 helloworld。如果看到了消息，就说明您在系统中成功地安装了这个程序。

▲

17.9 小结

本章介绍了有关命令行程序的基本知识以及标准输入、标准输出和标准错误之间的差别。您学习了如何使用 os 包来访问原始参数，以及如何使用 flag 包来分析命令行参数。学习了如何自定义帮助文本，以及如何在命令行工具中创建子命令。最后，您学习了如何安装和分享命令行工具。

17.10 问与答

问：如何查看命令的退出状态？

答：要查看命令的退出状态，在 Windows 系统中可使用 echo %errorlevel%，而在 macOS 和 Linux 系统中可使用 echo $?。请尝试运行一些命令，并查看其退出状态。

问：Go 语言为何将 -option 和 --option 视为同一个选项？

答：虽然使用一个连字符和两个连字符的选项通常是不同的选项，但 Go 设计者将它们视为相同的选项。

问：使用 go install 安装他人提供的命令行程序安全吗？

答：安装并运行包之前，务必检查其内容。在访问操作系统时，Go 程序的权限很大，因此务必谨慎地运行它们。想象一下，您安装了一些恶意代码，并试图做些研究来确定它是安全的。您明白了它是做什么的吗？这个程序被广泛使用吗？有您认识的人在使用它吗？

17.11 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看

后面的答案。

17.11.1 小测验

1．命令行程序出现错误时，如果您想将其显示给用户，应将其发送到哪种输出？

2．为何要使用 NewFlagSet？

3．请阐述 go get 和 go install 之间的差别。

17.11.2 答案

1．要在命令行脚本中显示错误，应使用标准错误。要将文本发送到标准错误，可像下面这样使用 fmt 和 os 包。

```
fmt.Fprintln(os.Stderr, "Something went wrong")
```

2．在命令行程序中，可使用方法 NewFlagSet 来创建子命令。您还可使用它来自定义错误处理行为。

3．go install 用于安装本地包，这可能是您编写的文件，也可能是您从网上或文件服务器中下载的文件。go install 从远程服务器（如 Github）获取文件，并像 go install 那样安装它们。这两个命令的作用大致相同，它们都安装文件，但 go get 还下载文件。

17.12 练习

1．查看您常用的一些程序的状态码。尝试触发错误，并查看状态码。

2．扩展程序 example05.go，给两个子命令都添加自定义帮助文本。在本书的代码示例中，文件 example06.go 提供了一种可能的解决方案。

3．编写一个简单的命令行程序，将其发布到 Github，并邀请朋友安装它。

第 18 章

创建 HTTP 服务器

本章介绍如下内容。

- 宣告 Web 服务器的存在。
- 查看请求和响应。
- 使用处理程序函数。
- 处理 404 错误。
- 设置报头。
- 响应以不同类型的内容。
- 响应不同类型的请求。
- 获取 GET 和 POST 请求中的数据。

Web 服务器可提供网页、Web 服务和文件，而 Go 语言为创建 Web 服务器提供了强大的支持。本章介绍如何创建 Web 服务器，使其能够响应不同的路由、不同类型的请求和不同类型的内容。使用 Go 语言编写 Web 服务器时，可利用 Go 语言提供的并发方法。

18.1 通过 Hello World Web 服务器宣告您的存在

标准库中的 net/http 包提供了多种创建 HTTP 服务器的方法，它还提供了一个基本路由器。传统上，通过创建一个 Hello World 程序来宣告您的存在。程序清单 18.1 是使用 Go 语言编写的最基本 HTTP 服务器。

程序清单 18.1 使用 Go 语言编写的基本 HTTP 服务器

```
1:  package main
2:
3:  import (
4:      "net/http"
5:  )
6:
```

```
 7:  func helloWorld(w http.ResponseWriter, r *http.Request) {
 8:      w.Write([]byte("Hello World\n"))
 9:  }
10:
11:  func main() {
12:      http.HandleFunc("/", helloWorld)
13:      http.ListenAndServe(":8000", nil)
14:  }
```

这个程序虽然只有 14 行，但做的工作却很多。

- 导入 net/http 包。
- 在 main 函数中，使用方法 HandleFunc 创建了路由/。这个方法接受一个模式和一个函数，其中前者描述了路径，而后者指定如何对发送到该路径的请求做出响应。
- 函数 helloWorld 接受一个 http.ResponseWriter 和一个指向请求的指针。这意味着在这个函数中，可查看或操作请求，再将响应返回给客户端。在这里，使用了方法 Write 来生成响应。这个方法生成的 HTTP 响应包含状态、报头和响应体。[ ]byte 声明一个字节切片并将字符串值转换为字节。这意味着方法 Write 可以使用[ ]byte，因为这个方法将一个字节切片作为参数。
- 为响应客户端，使用了方法 ListenAndServe 来启动一个服务器，这个服务器监听 localhost 和端口 8000。

这个示例很简单，但明白了如何使用 Go 语言编写 Web 服务器后，您就能创建更复杂的程序。

TRY IT YOURSELF ▼

运行 Web 服务器 Hello World

在这个示例中，您将运行 Web 服务 Hello World。

1. 在文本编辑器中打开文件 hour18/example01.go，并尝试理解这个示例是做什么的。如果需要，可参阅前面的说明，并详细阅读这些代码。
2. 在终端中，使用命令 go run example01.go 运行这个程序。
3. 启动 Web 浏览器，并访问 http://localhost:8000。
4. 您将在 Web 浏览器中看到 Hello World。
5. 祝贺您运行了第一个使用 Go 语言编写的 Web 服务器。

▲

18.2 查看请求和响应

在本章中，我们将查看发送请求和收到的响应。为此，我们将使用工具 curl。

curl 是一款用于发起 HTTP 请求的命令行工具，它几乎可在任何平台上使用。macOS 系统预先安装了 curl；Linux 系统通常也安装了 curl，如果没有安装，可使用包管理器进行安装；

Windows 系统没有预先安装 curl。对于 Windows 用户，本书前面已建议安装 GIT for Windows；如果您还没有安装，请访问 Git 官网下载安装程序，打开下载的文件并安装它。安装 GIT 后，“开始”菜单将包含选项 Git Bash，请打开它。

要核实正确地安装了所有所需的工具，请采取如下步骤。

1a．在 macOS 或 Linux 系统中，打开终端。

1b．在 Windows 系统中，通过“开始”菜单打开 Git Bash。

2．输入 curl 并按回车键。

3．如果成功地安装了 curl，您将看到如图 18.1 所示的输出。

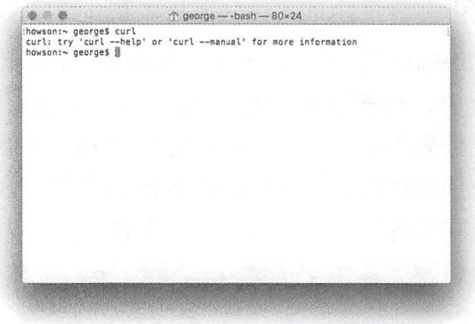


图 18.1 核实正确地安装了 curl

18.2.1 使用 curl 发出请求

安装 curl 后，就可在开发和调试 Web 服务器时使用它。您可不使用 Web 浏览器，而使用 curl 来向 Web 服务器发送各种请求以及查看响应。为向前面创建的 Hello World Web 服务器发出请求，请先打开终端并运行这个服务器。

```
go run example01.go
```

在 macOS 或 Linux 系统中，再打开一个终端；在 Windows 系统中，切换到 Git Bash。执行下面的命令，其中的选项 -is 指定打印报头，并忽略一些不感兴趣的内容。

```
curl -is http://localhost:8000
```

如果这个命令成功了，您将看到来自 Web 服务器的响应，其中包含报头和响应体。

```
HTTP/1.1 200 OK
Date:  Wed, 16 Nov 2016 16:45:51 GMT
Content-Length:  12
Content-Type:  text/plain; charset=utf-8

Hello World
```

对这些输出解读如下。

- 这个响应使用的协议为 HTTP 1.1，状态码为 200。
- 报头 Date 详细地描述了响应的发送时间。
- 报头 Content-Length 详细地指出了响应的长度，这里是 12 字节。

- 报头 Content-Type 指出了内容的类型以及使用的编码。在这里，响应的内容类型为 text/plain，是使用 utf-8 进行编码的。
- 最后输出的是响应体，这里是 Hello World。

18.2.2 详谈路由

HandleFunc 用于注册对 URL 地址映射进行响应的函数。简单地说，HandleFunc 创建一个路由表，让 HTTP 服务器能够正确地做出响应。

```
http.HandleFunc("/", helloWorld)
http.HandleFunc("/users/", usersHandler)
http.HandleFunc("/projects/", projectsHandler)
```

在这个示例中，每当用户向 / 发出请求时，都将调用函数 helloWorld，每当用户向 /users/ 发出请求时，都将调用函数 usersHandler，依此类推。

有关路由器的行为，有以下几点需要注意。

- 路由器默认将没有指定处理程序的请求定向到 /。
- 路由必须完全匹配。例如，对于向 /users 发出的请求，将定向到 /，因为这里末尾少了斜杆。
- 路由器不关心请求的类型，而只管将与路由匹配的请求传递给相应的处理程序。

18.3 使用处理程序函数

在 Go 语言中，路由器负责将路由映射到函数，但如何处理请求以及如何向客户端返回响应，是由处理程序函数定义的。很多编程语言和 Web 框架都采用这样的模式，即先由函数来处理请求和响应，再返回响应。在这方面，Go 语言也如此。处理程序函数负责完成如下常见任务。

- 读写报头。
- 查看请求的类型。
- 从数据库中取回数据。
- 分析请求数据。
- 验证身份。

处理程序函数能够访问请求和响应，因此一种常见的模式是，先完成对请求的所有处理，再将响应返回给客户端。响应生成后，就不能再对其做进一步的处理了。在下面的示例中，使用了方法 Write 来发送请求。接下来的一行设置了响应的一个报头。鉴于响应已经发送，这行代码不会有任何作用，但能够通过编译。这里要说的是，发送响应应是最后一步。

```
func helloWorld(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello World\n"))
    // This has no effect, as the response is already written
    w.Header().Set("X-My-Header", "I am setting a header!")
}
```

18.4 处理404错误

默认路由器的行为是将所有没有指定处理程序的请求都定向到/。回到第一个示例，如果用户请求的页面不存在，将调用给/指定的处理程序，从而返回响应Hello World和状态码200。

```
curl -is http://localhost:8000/asdfa
HTTP/1.1 200 OK
Date:  Thu, 17 Nov 2016 09:07:51 GMT
Content-Length:  12
Content-Type:  text/plain; charset=utf-8
```

然而，鉴于请求的路由不存在，原本应返回404错误（页面未找到）。为此，可在处理默认路由的函数中检查路径，如果路径不为/，就返回404错误，如程序清单18.2所示。

程序清单18.2 添加404响应

```
 1:  package main
 2:
 3:  import (
 4:      "net/http"
 5:  )
 6:
 7:  func helloWorld(w http.ResponseWriter, r *http.Request) {
 8:      if r.URL.Path != "/" {
 9:              http.NotFound(w, r)
10:              return
11:      }
12:      w.Write([]byte("Hello World\n"))
13:  }
14:
15:  func main() {
16:      http.HandleFunc("/", helloWorld)
17:      http.ListenAndServe(":8000", nil)
18:  }
```

相比于原来的Hello World Web服务器，这里所做的修改如下。

- 在处理程序函数helloWorld中，检查路径是否是/。
- 如果不是，就调用http包中的方法NotFound，并将响应和请求传递给它。这将向客户端返回一个404响应。
- 如果路径与/匹配，则if语句将被忽略，进而发送响应Hello World。

▼TRY IT YOURSELF

添加404响应

在这个示例中，您将明白如何添加404响应。

1. 在文本编辑器中打开文件hour18/example02.go，并尝试理解这个示例是做什么的。如果需要，请参阅前面的说明，并详细阅读代码。

2. 在终端中使用命令 go run example02.go 运行这个程序。
3. 使用 curl 请求一个不存在的页面，如以下代码所示。

```
curl -is http://localhost:8000/asdfa
```

4. 您看到的响应将为 404。

```
HTTP/1.1 404 Not Found
Content-Type:  text/plain; charset=utf-8
X-Content-Type-Options:  nosniff
Date:  Thu, 17 Nov 2016 09:15:23 GMT
Content-Length:  19

404 page not found
```

▲

18.5 设置报头

创建 HTTP 服务器时，经常需要设置响应的报头。在创建、读取、更新和删除报头方面，Go 语言提供了强大的支持。在下面的示例中，假设服务器将发送一些 JSON 数据。通过设置 Content-Type 报头，服务器可告诉客户端，发送的是 JSON 数据。处理程序函数可使用 ResponseWriter 来添加报头，如下所示。

```
w.Header().Set("Content-Type", "application/json; charset=utf-8")
```

只要这行代码是在响应被发送给客户端之前执行的，这个报头就会被添加到响应中。程序清单 18.3 中，在发送的 JSON 内容前添加了报头。请注意，为简单起见，将数据设置成了一个字符串，但数据通常是从别的地方读取的，并编码为 JSON 格式。

程序清单 18.3 在响应中添加报头

```
 1:  package main
 2:
 3:  import (
 4:      "net/http"
 5:  )
 6:
 7:  func helloWorld(w http.ResponseWriter, r *http.Request) {
 8:      if r.URL.Path != "/" {
 9:              http.NotFound(w, r)
10:              return
11:      }
12:      w.Header().Set("Content-Type", "application/json; charset=utf-8")
13:      w.Write([]byte(`{"hello":  "world"}`))
14:  }
15:
16:  func main() {
17:      http.HandleFunc("/", helloWorld)
18:      http.ListenAndServe(":8000", nil)
19:  }
```

TRY IT YOURSELF

在响应中包含 HTTP 报头

在这个示例中，您将明白如何添加 HTTP 报头。

1. 在文本编辑器中打开文件 hour18/example03.go，并尝试理解这个示例是做什么的。
2. 在终端中使用命令 go run example03.go 运行这个程序。
3. 使用 curl 向这个 Web 服务器发出请求。

```
curl -is http://localhost:8000/
```

4. 您将看到响应中包含一个设置内容类型的报头。

```
application/json.
HTTP/1.1 200 OK
Content-Type:  application/json; charset=utf-8
Date:  Thu, 17 Nov 2016 09:28:44 GMT
Content-Length:  18

{"hello":  "world"}
```

5. 祝贺您使用 Go 语言提供了 JSON 数据！

18.6 响应以不同类型的内容

响应客户端时，HTTP 服务器通常提供多种类型的内容。一些常用的内容类型包括 text/plain、text/html、application/json 和 application/xml。如果服务器支持多种类型的内容，客户端可使用 Accept 报头请求特定类型的内容。这意味着同一个 URL 可能向浏览器提供 HTML，而向 API 客户端提供 JSON。只需对本章的示例稍作修改，就可让它查看客户端发送的 Accept 报头，并据此提供不同类型的内容，如程序清单 18.4 所示。

程序清单 18.4 以不同类型的内容进行响应

```
 1:  package main
 2:
 3:  import (
 4:      "net/http"
 5:  )
 6:
 7:  func helloWorld(w http.ResponseWriter, r *http.Request) {
 8:      if r.URL.Path != "/" {
 9:              http.NotFound(w, r)
10:              return
11:      }
12:      switch r.Header.Get("Accept") {
13:      case "application/json":
14:              w.Header().Set("Content-Type", "application/json; charset=utf-8")
15:              w.Write([]byte(`{"message":  "Hello World"}`))
16:      case "application/xml":
17:              w.Header().Set("Content-Type", "application/xml; charset=utf-8")
18:              w.Write([]byte(`<?xml version="1.0" encoding="utf-
    8"?><Message>Hello World</Message>`)
```

```
19:      default:
20:              w.Header().Set("Content-Type", "text/plain; charset=utf-8")
21:              w.Write([]byte("Hello World\n"))
22:      }
23:
24:  }
25:
26:  func main() {
27:      http.HandleFunc("/", helloWorld)
28:      http.ListenAndServe(":8000", nil)
29:  }
```

对这里所做的扩展解读如下。

- 在函数 helloWorld 中，添加了一条 switch 语句，用户检查客户端发送的 Accept 报头。
- 这条 switch 语句根据 Accept 报头的内容相应地设置响应。
- 如果没有找到这个报头，服务器默认将发送 text/plain 响应。

TRY IT YOURSELF ▼

提供不同类型的内容

在这个示例中，您将明白如何提供和请求不同类型的内容。

1. 在文本编辑器中打开文件 hour18/example04.go，并尝试理解这个示例是做什么的。如果需要，可参阅前面的说明并详细阅读代码。

2. 在终端中使用命令 go run example04.go 运行这个程序。

3. 使用 curl 向这个 Web 服务器发出请求，并将内容类型设置为 application/json。请注意，要设置报头，可使用选项 -H。

```
curl -si -H 'Accept:  application/json' http://localhost:8000
```

4. 您将看到，响应的内容类型为 application/json。

```
HTTP/1.1 200 OK
Content-Type:  application/json; charset=utf-8
Date:  Thu, 17 Nov 2016 09:28:44 GMT
Content-Length:  18

{"hello":  "world"}
```

5. 使用 curl 向这个 Web 服务器发出请求，并将内容类型设置为 application/xml。

```
curl -si -H 'Accept:  application/xml' http://localhost:8000
```

6. 您将看到响应的内容类型为 application/xml。

```
HTTP/1.1 200 OK
Content-Type:  application/xml; charset=utf-8
Date:  Thu, 17 Nov 2016 09:45:24 GMT
Content-Length:  68

<?xml version="1.0" encoding="utf-8"?><Message>Hello World</Message>
```

▲

18.7 响应不同类型的请求

除响应以不同类型的内容外，HTTP 服务器通常也需要能够响应不同类型的请求。客户端可发出的请求类型是 HTTP 规范中定义的，包括 GET、POST、PUT 和 DELETE。要使用 Go 语言创建能够响应不同类型请求的 HTTP 服务器，可采用类似于提供多种类型内容的方法，如程序清单 18.5 所示。在路由的处理程序函数中，可使用 switch 语句检查请求类型，并决定如何处理请求。

程序清单 18.5 响应不同类型的请求

```
package main

import (
    "net/http"
)

func helloWorld(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/" {
            http.NotFound(w, r)
            return
    }
    switch r.Method {
    case "GET":
            w.Write([]byte("Received a GET request\n"))
    case "POST":
            w.Write([]byte("Received a POST request\n"))
    default:
            w.WriteHeader(http.StatusNotImplemented)
            w.Write([]byte(http.StatusText(http.StatusNotImplemented)) + "\n")
    }

}

func main() {
    http.HandleFunc("/", helloWorld)
    http.ListenAndServe(":8000", nil)
}
```

这里对 Web 服务器所做的扩展如下。

- 服务器不是根据内容类型来生成响应，而是根据请求方法来生成响应。
- switch 语句根据请求类型发送不同的响应。
- 在这个示例中，发送一个 plain/text 响应来指出请求的类型。
- 如果请求方法不是 GET 或 POST，就执行 default 子句，发送 501（Not Implemented HTTP）响应。代码 501 意味着服务器不明白或不支持客户端使用的 HTTP 请求方法。

如果您运行这个服务器，将发现现在可以发出 GET 或 POST 请求。要使用 curl 来修改请求类型，可使用选项 -X。

TRY IT YOURSELF ▼

理解不同类型的请求

在这个示例中，您将明白如何响应不同类型的请求，如 GET 和 POST。

1. 在文本编辑器中打开文件 hour18/example05.go，并尝试理解这个示例是做什么的。如果需要，可参阅程序清单 18.5 后面的说明。

2. 在终端中使用命令 go run example05.go 运行这个程序。

3. 使用 curl 向这个 Web 服务器发出 GET 请求。请注意，要设置请求的类型，可使用选项 -X。

```
curl -si -X GET http://localhost:8000
```

4. 您将看到服务器在响应中指出它收到了一个 GET 请求。

```
HTTP/1.1 200 OK
Date:  Thu, 17 Nov 2016 10:02:49 GMT
Content-Length:  23
Content-Type:  text/plain; charset=utf-8

Received a GET request
```

5. 再次使用 curl 向这个 Web 服务器发出请求，但这次发出的是 POST 请求。

```
curl -si -X POST http://localhost:8000
```

6. 您将看到服务器在响应中指出它收到了一个 POST 请求。

```
HTTP/1.1 200 OK
Date:  Thu, 17 Nov 2016 10:03:27 GMT
Content-Length:  24
Content-Type:  text/plain; charset=utf-8

Received a POST request
```

▲

18.8 获取 GET 和 POST 请求中的数据

HTTP 客户端可在 HTTP 请求中向 HTTP 服务器发送数据，这样的典型示例包括以下几点。

- 提交表单。
- 设置有关要返回的数据的选项。
- 通过 API 管理数据。

在 Go 语言中，获取客户端请求中的数据很简单，但获取方式随请求类型而异。对于 GET 请求，其中的数据通常是通过查询字符串设置的。一个通过 GET 请求发送数据的例子是，在 Google 网站上搜索。在这个示例中，URL 包含以查询字符串的方式指定的搜索关键字。

```
https://www.google.com/?q=golang
```

Web服务器可能读取查询字符串中的数据，并使用它来做些事情，如从数据库获取相应的数据，再将其返回给客户端。在Go语言中，以字符串映射的方式提供了请求中的查询字符串参数，您可使用range子句来遍历它们。

```
func queryParams(w http.ResponseWriter, r *http.Request) {
    for k, v := range r.URL.Query() {
        fmt.Printf("%s:  %s\n", k, v)
    }
}
```

在POST请求中，数据通常是在请求体中发送的。要读取并使用这些数据，可像下面这样做。

```
func queryParams(w http.ResponseWriter, r *http.Request) {
    reqBody, err := ioutil.ReadAll(r.Body)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s", reqBody)
}
```

Watch Out!

警告： 可别信任用户的输入。

这里简单地说一说安全问题。在服务器上，应将收到的数据视为不可信任的。攻击者可能发送请求，企图窃取信息、获取对服务器的访问权或删除数据库。所有进入服务器的数据都应视为不安全的，应过滤后再使用。本章的这些示例中，在未经过滤的情况下就直接使用了收到的数据，但编写用于生产环境的代码时，务必确保接收的数据没问题后再使用。

现在可以创建一个完整的代码示例，以演示如何处理来自不同请求的数据了，如程序清单18.6所示。这个服务器是在前一个示例的基础上创建的，旨在显示发送给服务器的数据。如果您运行这个示例，将发现数据可能来自不同类型的请求。当然，服务器可能对数据做更有趣的处理，而不仅仅是将其返回给客户端。

程序清单18.6 处理不同请求中的数据

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "io/ioutil"
 6:      "log"
 7:      "net/http"
 8:  )
 9:
10:  func helloWorld(w http.ResponseWriter, r *http.Request) {
11:      if r.URL.Path != "/" {
12:              http.NotFound(w, r)
13:              return
14:      }
15:      switch r.Method {
16:      case "GET":
17:              for k, v := range r.URL.Query() {
18:                      fmt.Printf("%s:  %s\n", k, v)
```

```
19:             }
20:             w.Write([]byte("Received a GET request\n"))
21:     case "POST":
22:             reqBody, err := ioutil.ReadAll(r.Body)
23:             if err != nil {
24:                     log.Fatal(err)
25:             }
26:
27:             fmt.Printf("%s\n", reqBody)
28:             w.Write([]byte("Received a POST request\n"))
29:     default:
30:             w.WriteHeader(http.StatusNotImplemented)
31:             w.Write([]byte(http.StatusText(http.StatusNotImplemented)))
32:     }
33:
34: }
35:
36: func main() {
37:     http.HandleFunc("/", helloWorld)
38:     http.ListenAndServe(":8000", nil)
39: }
```

TRY IT YOURSELF ▼

获取 GET 和 POST 请求中的数据

在这个示例中，您将明白如何获取 GET 和 POST 请求中的数据。

1. 在文本编辑器中打开文件 hour18/example06.go，并尝试理解这个示例是做什么的。
2. 在终端中使用命令 go run example06.go 运行这个程序。
3. 使用 curl 向这个 Web 服务器发出包含一些查询参数的 GET 请求。

```
curl -si "http://localhost:8000/?foo=1&bar=2"
```

4. 在运行服务器的终端中，您将看到服务器收到了这些数据。

```
foo: [1]
bar: [2]
```

5. 再次使用 curl 向这个 Web 服务器发出请求，但这次发出的是 POST 请求。

```
curl -si -X POST -d "some data to send" http://localhost:8000/
```

6. 在运行服务器的终端中，您将看到服务器收到了这些数据。

```
some data to send
```

▲

18.9 小结

本章介绍了如何使用 Go 语言创建 HTTP 服务器。首先，介绍了 http 包中的路由原理，

以及如何使用处理程序函数来处理请求；接着，您学习了如何设置 HTTP 响应的报头，以及如何响应不同类型的请求；最后，您学习了如何获取 HTTP 客户端请求中的数据。

18.10 问与答

问：在路由模式中，可使用变量吗？我想使用类似于/products/:id 这样的模式，其中:id 是一个变量。

答：http 包默认使用 ServeMux 来处理路由，因此不支持变量，也不支持正则表达式。社区创建的一些流行的路由器支持变量以及请求和内容类型等特性。一般而言，它们能够与 http 包集成起来。

问：我使用过其他语言的框架库来创建服务器，Go 语言提供了类似的东西吗？

答：是的，也有 Go 语言框架库。然而，在大多数情况下，http 包提供了您需要的所有功能。

问：如何创建 HTTPS 服务器？

答：http 包提供了方法 ListenAndServeTLS，可用来创建 HTTPS（TLS）服务器。这个方法的工作原理与 ListenAndServe 相同，但必须向它传递证书和密钥文件。

18.11 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

18.11.1 小测验

1．HTTP GET 请求和 POST 请求有何不同？

2．在处理程序函数中，w.Write 在处理响应方面做了什么（其中 w 是一个 ResponseWriter）？

3．该信任用户提交给 HTTP 服务器的数据吗？

18.11.2 答案

1．GET 请求向指定资源请求数据，而 POST 请求向指定资源提交数据。GET 请求可能通过查询字符串参数发送数据，而 POST 请求通过消息体向服务器发送数据。

2．对 ResponseWriter 调用 Write 导致响应被发送给客户端，这包括报头和响应体的内容。响应一旦发送，就无法对其进行修改。

3．不应该。绝不要信任客户端提交给服务器的数据，应确定数据没问题后再使用。

18.12 练习

1．修改 example04.go，使其能够在收到 HTML 请求时响应以一个简单的 HTML 文档。HTML 内容类型为 "text/html; charset=utf-8"。

2．修改 example05.go，让这个服务器支持 DELETE 请求。

3．阅读有关 http 包的文档，尝试理解编写服务器时可对请求和响应做的一些修改。

第 19 章

创建 HTTP 客户端

本章介绍如下内容。

- 理解 HTTP。
- 发出 GET 请求。
- 发出 POST 请求。
- 进一步控制 HTTP 请求。
- 调试 HTTP 请求。
- 处理超时。

超文本传输协议（Hypertext Transfer Protocol，HTTP）是一种在互联网上收发资源的网络协议，用于传输图像、HTML 文档和 JSON 等。本章介绍如何使用 Go 语言创建 HTTP 客户端，您将学习如何发出各种类型的请求，还有如何在开发期间调试程序。

19.1 理解 HTTP

要理解 HTTP 请求的结构，一种不错的方式是使用 curl。您在第 18 章创建 HTTP 服务器时使用过 curl，它在您开发 HTTP 客户端时也很有用。通过下面的命令获取了 Google 主页，但请不要过度关注这个命令及其语法。这里应关注的是 HTTP 请求的结构。

```
$ curl -s -o /dev/null -v http://google.com
> GET / HTTP/1.1
> Host:  google.com
> User-Agent:  curl/7.43.0
> Accept:  */*
>
< HTTP/1.1 302 Found
< Cache-Control:  private
< Content-Type:  text/html; charset=UTF-8
< Referrer-Policy:  no-referrer
< Location:  http://www.google.co.uk/?gfe_rd=cr&ei=ALMhWdzRK4qwcpmaoLAE
```

```
< Content-Length:  259
< Date:  Sun, 21 May 2017 15:32:16 GMT
<
{ [259 bytes data]
* Connection #0 to host google.com left intact
```

这个在 verbose 模式下的 curl 描述了服务器和客户端之间传输的多个请求响应。其中的日志详细说明了发出的请求和收到的响应，对其解读如下。

- 以字符 > 打头的行描述了客户端发出的请求。
- 以字符 < 打头的行描述收到的响应。
- 请求的详细信息描述了随请求发送的一些报头，这些报头向服务器提供了一些有关客户端的信息。
- 响应详细描述了一些报头，这些报头指出了响应的内容类型、长度和发送时间。

如果您没有安装 curl，也可通过 Google Chrome 开发者工具来了解 HTTP 请求的特征。

1. 打开 Web 浏览器 Google Chrome。
2. 打开 Chrome 开发者工具，它能够让您查看下载网页时浏览器在幕后所做的工作。
3. 在地址栏中输入 BBC 主页网址并按回车键，以访问 BBC 网站。
4. 加载这个网页时，Network 选项卡中将显示大量的 HTTP 请求和响应。
5. 单击 Name 栏中的任何条目。
6. 将打开一系列选项卡，并显示 Headers 选项卡。这个选项卡详细描述了访问该网页时发出的一个 HTTP 请求及其响应的报头。
7. 研究请求和响应的报头，您能识别其中表示内容类型或长度的报头吗？

这里介绍 curl 和 Chrome 开发者工具旨在指出，要创建 HTTP 客户端，必须对 HTTP 请求的结构有大致认识。Go 语言中的 HTTP 客户端功能齐备，在最基本的情况下，它给选项提供了合理的默认设置，让开发人员无须关心它们。Go 语言的 HTTP 客户端支持细粒度的控制，因此对 HTTP 有深入认识是有必要的。

如果您的目标不仅仅是与 HTTP 进行简单交互，就必须对 HTTP 规范有更深入的认识。

19.2 发出 GET 请求

Go 语言在 net/http 包中提供了一个快捷方法，可用于发出简单的 GET 请求。使用这个方法意味着不需要考虑如何配置 HTTP 客户端以及如何设置请求报头。如果只是要从远程网站获取一些数据，那么默认配置完全够用。

在程序清单 19.1 中，客户端访问网站 ifconfig.io 的主页。这个网站报告请求网页客户端的 IP 地址。

程序清单 19.1　GET 请求

```
1:  package main
2:
3:  import (
4:      "fmt"
```

```
    "io/ioutil"
    "log"
    "net/http"
)

func main() {
    response, err := http.Get("https://ifconfig.io/")
    if err != nil {
        log.Fatal(err)
    }
    defer response.Body.Close()
    body, err := ioutil.ReadAll(response.Body)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s", body)
}
```

对这个示例解读如下。

- 使用 net/http 包向 https://ifconfig.io/ 发出 GET 请求。
- 如果请求出错（如没有网络连接），就将错误写入日志再退出。
- 客户端读取所有数据后，将连接关闭。
- 将响应体读取到变量中以便打印。
- 如果读取响应体时出错，就将错误写入日志再退出。
- 打印响应体。

如果您执行这个程序，则将向 ifconfig.io 发出请求，并返回发出请求的客户端的 IP 地址。

```
$ go run example01.go
68.235.53.83
```

▼ TRY IT YOURSELF

向 Web 服务发出 GET 请求

在这个示例中，您将向一个 Web 服务发出 GET 请求，以获悉您的外部 IP 地址。

1. 在文本编辑器中打开文件 hour19/example01.go，并尝试理解这个示例是做什么的。如果需要，可参阅前面的说明并详细阅读代码。

2. 在终端中使用命令 go run example01.go 运行这个程序。

3. 您将在终端中看到您的外部 IP 地址。

▲

19.3 发出 POST 请求

标准库中的 net/http 包也提供了用于发出简单 POST 请求的快捷方法——Post，它支持设

置内容类型以及发送数据。

程序清单 19.2 所示的客户端向 https://httpbin.org/post 发出 POST 请求。httpbin 是一个用于测试 HTTP 客户端的工具，而端点 /post 返回客户端发送给它的数据，以及有关客户端的信息。

程序清单 19.2 POST 请求

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "io/ioutil"
 6:      "log"
 7:      "net/http"
 8:      "strings"
 9:  )
10:
11:  func main() {
12:      postData := strings.NewReader(`{ "some":  "json" }`)
13:      response, err := http.Post("https://httpbin.org/post", "application/json",
    postData)
14:      if err != nil {
15:          log.Fatal(err)
16:      }
17:      defer response.Body.Close()
18:      body, err := ioutil.ReadAll(response.Body)
19:      if err != nil {
20:          log.Fatal(err)
21:      }
22:      fmt.Printf("%s", body)
23:  }
```

对这个示例解读如下。

- 声明变量 postData 并将一个 JSON 字符串赋给它。这里使用了标准库 strings 将其转换为 io.Reader，为传输做好准备。
- 使用方法 Post 发出 POST 请求。其中第一个参数是要发送到的 URL，第二个为数据的内容类型，第三个为数据本身。
- 如果请求出错（如没有网络连接），就将错误写入日志再退出。
- 客户端读取所有数据后，将连接关闭。
- 将响应体读取到变量中以便打印。
- 如果读取响应体时出错，就将错误写入日志再退出。
- 打印响应体。

如果您运行这个程序，从 httpbin 返回的响应将指出成功地以 JSON 方式发送了数据。您可查看发送的原始数据，以及服务器分析得到的数据。

```
$ go run example02.go
{
  "args":  {},
  "data":  "{ \"some\":  \"json\" }",
  "files":  {},
  "form":  {},
  "headers":  {
```

```
    "Accept-Encoding": "gzip",
    "Connection": "close",
    "Content-Length": "18",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "Go-http-client/1.1"
  },
  "json": {
    "some": "json"
  },
  "origin": "68.235.53.83",
  "url": "https://httpbin.org/post"
}
```

19.4 进一步控制 HTTP 请求

要进一步控制 HTTP 请求，应使用自定义的 HTTP 客户端。您可使用 net/http 包提供的默认 HTTP 客户端，但这将自动使用默认设置，除非您手工修改这些设置。程序清单 19.3 与程序清单 19.1 等价，但使用的是设置为默认的自定义 HTTP 客户端。

程序清单 19.3 使用自定义客户端

```
 1: package main
 2:
 3: import (
 4:     "fmt"
 5:     "io/ioutil"
 6:     "log"
 7:     "net/http"
 8: )
 9:
10: func main() {
11:     client := &http.Client{}
12:     request, err := http.NewRequest("GET", "https://ifconfig.co", nil)
13:     if err != nil {
14:         log.Fatal(err)
15: }
16:
17:     response, err := client.Do(request)
18:     defer response.Body.Close()
19:     body, err := ioutil.ReadAll(response.Body)
20:     if err != nil {
21:         log.Fatal(err)
22:     }
23:     fmt.Printf("%s", body)
24: }
```

对为使用自定义 HTTP 客户端所做的修改解读如下。

- 不使用 net/http 包的快捷方法 Get，而创建一个 HTTP 客户端。
- 使用方法 NewRequest 向 https://ifconfig.co 发出 GET 请求。
- 使用方法 Do 发送请求并处理响应。

如果您运行这个程序，结果将与前一个示例相同。

```
$ go run example03.go
68.235.53.83
```

使用自定义 HTTP 客户端意味着可对请求设置报头、基本身份验证和 cookies。鉴于使用快捷方法和自定义 HTTP 客户端时，发出请求所需代码的差别很小，建议除非要完成的任务非常简单，否则都使用自定义 HTTP 客户端。

19.5 调试 HTTP 请求

创建 HTTP 客户端时，了解收发请求和响应的报头和数据对整个流程很有用。为此，可使用标准库中的 fmt 包来输出各项数据，但 net/http/httputil 也提供了能够让您轻松调试 HTTP 客户端和服务器的方法。这个包中的方法 DumpRequestOut 和 DumpResponse 能够让您查看请求和响应。

可对前一个示例进行改进，以使用 net/http/httputil 包中的 DumpRequestOut 和 DumpResponse 方法来支持日志功能。这些方法显示请求和响应的报头，还有返回的响应体。

可在调试时添加这些方法，并在调试完毕后删除它们，但还有一种选择，那就是使用环境变量来开关调试。标准库中的 os 包支持读取环境变量，这能够让您轻松地开关调试。程序清单 19.4 演示了如何调试 HTTP GET 请求。

程序清单 19.4 调试请求

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "io/ioutil"
 6:      "log"
 7:      "net/http"
 8:      "net/http/httputil"
 9:      "os"
10:  )
11:
12:  func main() {
13:      debug := os.Getenv("DEBUG")
14:      client := &http.Client{}
15:      request, err := http.NewRequest("GET", "https://ifconfig.co", nil)
16:      if err != nil {
17:          log.Fatal(err)
18:      }
19:
20:      if debug == "1" {
21:          debugRequest, err := httputil.DumpRequestOut(request, true)
22:          if err != nil {
23:              log.Fatal(err)
24:          }
25:          fmt.Printf("%s", debugRequest)
26:      }
27:      response, err := client.Do(request)
28:      defer response.Body.Close()
29:
30:      if debug == "1" {
31:          debugResponse, err := httputil.DumpResponse(response, true)
32:          if err != nil {
33:              log.Fatal(err)
34:          }
35:          fmt.Printf("%s", debugResponse)
36:      }
37:      body, err := ioutil.ReadAll(response.Body)
```

```
38:       if err != nil {
39:           log.Fatal(err)
40:       }
41:
42:       fmt.Printf("%s\n", body)
43:   }
```

如果在启用了调试的情况下执行这个程序，将显示请求和响应的报头，以及响应体。请注意，这个程序打印两次响应体，因为调试信息中也包含响应体。

```
# For OSX and Linux
$ DEBUG=1 go run example04.go
# For Windows
$ set DEBUG=1
$ go run example04.go

GET / HTTP/1.1
Host:  ifconfig.co
User-Agent:  Go-http-client/1.1
Accept-Encoding:  gzip

HTTP/1.1 200 OK
Content-Length:  12
Connection:  keep-alive
Content-Type:  text/plain; charset=utf-8
Date:  Tue, 15 Nov 2016 09:48:04 GMT
Server:  nginx
Strict-Transport-Security:  max-age=31536000; includeSubdomains; preload

68.235.53.83
68.235.53.83
```

现在的输出包含有关请求和响应的丰富信息，您能够通过报头和数据知道向服务器发送的是什么，以及服务器返回的是什么。创建HTTP客户端时，这些信息很有用，它们可帮助您查找Bug以及了解没有收到响应的原因。例如，假设您想要收到JSON格式的响应，但调试信息表明服务器返回的内容类型为text/plain。您查看请求报头，会发现这里没有指出客户端需要哪种类型的数据，因此服务器以纯文本的方式返回数据就没什么可奇怪的了。

要请求JSON数据，可修改客户端，设置一个请求JSON的报头。为此，可像下面这样做。

```
request.Header.Add("Accept", "application/json")
```

现在如果您发出请求，将发现请求的是JSON数据，且成功地返回了这样的数据。

```
# For macOS and Linux
DEBUG=1 go run example05.go
# For Windows
set DEBUG=1
go run example05.go

GET / HTTP/1.1
Host:  ifconfig.co
User-Agent:  Go-http-client/1.1
Accept:  application/json
Accept-Encoding:  gzip

HTTP/1.1 200 OK
Transfer-Encoding:  chunked
Connection:  keep-alive
```

```
Content-Type:  application/json
Date:  Tue, 15 Nov 2016 10:33:48 GMT
Server:  nginx
Strict-Transport-Security:  max-age=31536000; includeSubdomains; preload
Vary:  Accept-Encoding

88
{"ip":"68.235.53.83","ip_decimal":1156265299,"country":"United States","city":
"Chicago","hostname":"static-68-235-53-83.cust.tzulo.com"}
0

{"ip":"68.235.53.83","ip_decimal":1156265299,"country":"United States","city":
"Chicago","hostname":"static-68-235-53-83.cust.tzulo.com"}%
```

通过使用 Accept 报头，客户端告诉服务器它想要的是 application/json，而服务器返回数据时将 Content-Type 报头设置成了 application/json。虽然大多数第三方 API 都遵循了 HTTP 规范，但您使用的 API 可能会导致实现不正确，或者不支持 JSON。在这种情况下，使用 Go 语言调试功能可快速发现 Bug 和问题。

19.6 处理超时

HTTP 事务会为接收响应等待一定的时间。客户端向服务器发送请求后，完全无法知道响应会在多长时间内返回。在底层，有大量影响响应速度的变数。

- DNS 查找速度。
- 打开到服务器 IP 地址的 TCP 套接字的速度。
- 建立 TCP 连接的速度。
- TLS 握手的速度（如果连接是 TLS 的）。
- 向服务器发送数据的速度。
- 重定向的速度。
- Web 服务器返回响应的速度。
- 将数据传输到客户端的速度。

如果您不明白这些阶段的含义，也不用担心。您只需知道 HTTP 响应的速度无法预测就够了。例如，向同一个 Web 服务器发出请求时，一次可能需要 1000 ms，而另一次可能需要 10000ms。对 HTTP 客户端来说，这是一个问题，因为每条连接都会占用一些内存，还会使用底层操作系统中的一个套接字。如果连接的速度很慢，则程序很快就会出现内存泄露，或耗尽底层操作系统的资源。

使用默认的 HTTP 客户端时，没有对请求设置超时时间。这意味着如果服务器没有响应，则请求将无限期地等待或挂起。对于任何请求，都建议设置超时时间。这样如果请求在指定的时间内没有完成，将返回错误。

```
client := &http.Client{
    Timeout:  1 * time.Second,
}
```

上述配置要求客户端在 1s 内完成请求。为演示这一点，可在前面的示例中设置超时时间。

这里设置的超时时间很短，因此完全可能出现超时的情况。

```
client := &http.Client{
    Timeout:  50 * time.Millisecond,
}
```

运行这个程序将导致错误，因为服务器的响应速度不够快。

```
$ go run example06.go
Get https://ifconfig.io:  net/http:  request canceled while waiting for connection
(Client.Timeout exceeded while awaiting headers)
exit status 1
```

▼ TRY IT YOURSELF

设置超时时间

在这个示例中，您将设置HTTP请求的超时时间。

1. 在文本编辑器中打开文件hour19/example06.go，并尝试理解这个示例是做什么的。
2. 在终端中使用命令go run example06.go运行这个程序。
3. 您将在终端中看到错误。

▲

通过创建一个传输（transport）并将其传递给客户端，可更细致地控制超时：控制HTTP连接的各个阶段。在大多数情况下，使用Timeout就足以控制整个HTTP事务，但在Go语言中，还可通过创建传输来控制HTTP事务的各个部分。

```
tr := &http.Transport{
    DialContext:  (&net.Dialer{
        Timeout:    30 * time.Second,
        KeepAlive:  30 * time.Second,
    }).DialContext,
    TLSHandshakeTimeout:   10 * time.Second,
    IdleConnTimeout:       90 * time.Second,
    ResponseHeaderTimeout: 10 * time.Second,
    ExpectContinueTimeout: 1 * time.Second,
}

client := &http.Client{
    Transport:  tr,
}
```

19.7 小结

本章介绍了如何使用Go语言创建HTTP服务器。首先，介绍了http包中的路由原理，以及如何使用处理程序函数来处理请求，接着，您学习了如何设置HTTP响应的报头，以及如何响应不同类型的请求；最后，您学习了如何获取HTTP客户端请求中的数据。

19.8 问与答

问：我对 HTTP 不熟悉，感觉它难以理解。请问这种感觉正常吗？

答：HTTP 的细节可能难以理解，在您没有系统学习计算机时尤其如此。如果您刚接触 Go 语言和开发，只需知道 HTTP 定义了客户端和服务器交互的一种方式就够了。通过使用 Go 语言提供的快捷方法，学习 HTTP 编程将会变得很容易。等您创建更多的 HTTP 客户端和服务器后，就会对 HTTP 更熟悉。如果您要创建使用 HTTP 的服务，花点时间去了解 HTTP 是值得的。要熟悉 HTTP，必须阅读 HTTP 规范。

问：能够同时发出多个 HTTP 请求吗？

答：可以。通过使用 goroutine，客户端可同时发出多个 HTTP 请求。

问：能够根据返回 HTTP 状态码调整程序采取的措施吗？

答：可以。可通过 Response.StatusCode 来访问响应的状态码，因此可编写基于服务器响应的逻辑。

19.9 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

19.9.1 小测验

1. 报头 Accept 和 Content-Type 是做什么的？
2. 如何给请求设置自定义报头？
3. 什么情况下该使用 net/http 包中的方法 Get 和 Post？什么情况下该使用 NewRequest？

19.9.2 答案

1. 报头 Accept 告诉服务器，客户端能够接收哪些类型的内容。报头 Content-Type 是服务器发送的，它指出当前发送给客户端的是哪种类型的数据。如果客户端使用报头 Accept 向服务器请求 application/json，而服务器也支持这种类型，就应在返回数据时将报头 Content-Type 设置为 application/json。

2. 如果创建了 HTTP 客户端，可像下面这样设置请求的报头。

```
client := &http.Client{}
 request, err := http.NewRequest("GET", "http://www.XXX.com", nil)
 request.Header.Add("Connection", "close")
```

3. 如果您对 HTTP 不熟悉，可使用方法 Get 和 Post 来快速实现 HTTP 编程。使用这些方法时，还可创建 HTTP 客户端以及指定超时时间，如下所示。

```
client := &http.Client{
          Timeout:  1 * time.Second,
}
resp, err := client.Get("http://XXX.com")
```

如果需要控制报头和请求的其他方面，应使用方法 NewRequest。

19.10 练习

1．创建一个 HTTP 客户端，向 https://google.com/404 发出 GET 请求，并将响应状态码打印出来。状态码是 404 吗？

2．创建一个 HTTP 客户端，向 https://httpbin.org/post 发出 POST 请求以发送一些数据，再检查响应看一看是否成功地发送了数据。

3．创建一个 HTTP 客户端，向 https://httpbin.org/user-agent 发出 GET 请求，并将其报头 User-Agent 设置为 GolangBot。检查响应中的用户代理（user-agent）值，它是您在报头 User Agent 设置的 GolangBot 吗？

第 20 章

处理 JSON

本章介绍如下内容。

- JSON 简介。
- 使用 JSON API。
- 在 Go 语言中使用 JSON。
- JSON 解码。
- 映射数据类型。
- 处理通过 HTTP 收到的 JSON。

本章介绍如何在 Go 语言中处理 JSON。您将学习如何对 JSON 进行编码和解码，还有 JavaScript 和 Go 数据类型之间的一些不同。您将明白如何使用结构体标签（struct tags）细致地控制 JSON，还有如何使用 HTTP 从远程 API 取回 JSON。

20.1 JSON 简介

JavaScript 对象表示法（JavaScript Object Notation，JSON）是一种用于存储和交换数据的格式，这是一种人类能够理解的纯文本格式。JSON 可以键值对的方式表示数据，也可以数组的方式表示数据。JSON 最初是一个 JavaScript 子集，但它现在已独立于语言，实际上，大多数语言都支持 JSON 数据编码和解码。JSON 已成为互联网上存储和交换数据的事实标准，从很大程度上说它已取代可扩展的标记语言（Extensible Markup Language，XML）。虽然很多现代数据服务依然提供 XML 格式，但 JSON 已经是互联网上最常见的数据格式。如果您所做的编程工作涉及使用或提供互联网服务，那么很可能需要用到 JSON。

程序清单 20.1 显示的响应是由处理资源 GET /user/:username 的 Github AP 提供的，这些数据是以文本格式提供的，客户端可随意使用。目标语言（如 JavaScript、Ruby、C#）使用这些数据前，很可能先对其进行分析。

程序清单 20.1 JSON 对象示例

```
 1: {
 2:     "login": "octocat",
 3:     "id": 1,
 4:     "avatar_url": "https://github.com/images/error/octocat_happy.gif",
 5:     "gravatar_id": "",
 6:     "url": "https://api.github.com/users/octocat",
 7:     "html_url": "https://github.com/octocat",
 8:     "followers_url": "https://api.github.com/users/octocat/followers",
 9:     "following_url": "https://api.github.com/users/octocat/following{/
    other_user}",
10:     "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
11:     "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/
    repo}",
12:     "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
13:     "organizations_url": "https://api.github.com/users/octocat/orgs",
14:     "repos_url": "https://api.github.com/users/octocat/repos",
15:     "events_url": "https://api.github.com/users/octocat/events{/privacy}",
16:     "received_events_url": "https://api.github.com/users/octocat/
    received_events",
17:     "type": "User",
18:     "site_admin": false,
19:     "name": "monalisa octocat",
20:     "company": "GitHub",
21:     "blog": "https://github.com/blog",
22:     "location": "San Francisco",
23:     "email": "octocat@github.com",
24:     "hireable": false,
25:     "bio": "There once was...",
26:     "public_repos": 2,
27:     "public_gists": 1,
28:     "followers": 20,
29:     "following": 0,
30:     "created_at": "2008-01-14T04:33:35Z",
31:     "updated_at": "2008-01-14T04:33:35Z"
32: }
```

程序清单 20.1 包含的都是键值对，但键对应的值也可以是数组，如下例所示。

```
{
    "name":"George",
    "age":40,
    "children":[ "Bea", "Fin"]
}
```

虽然 Go 语言也支持数组，但更常见的是使用切片来表示一组元素。在其他编程语言中，数组也被称为列表，尽管这有点令人迷惑，但数组和列表的定义相同，都是一组元素。

JSON 得以流行是因为它是一种人类能够看懂的灵活而轻量级的数据格式。虽然 XML 提供了模式（严格的数据表示方式），但程序员完全可以随心所欲地表示数据。从表示数据所需的字节数上说，JSON 通常更为轻量级。通过诸如互联网等网络发送数据时，可能意味着应用程序的运行速度稍快。另外，JavaScript 是占统治地位的 Web 浏览器编程环境，而 JSON 是 JavaScript 的一个子集，因此编码和解码 JSON 数据就是小菜一碟。

20.2 使用 JSON API

近年来，互联网上出现了很多卓越的 JSON API。现在，通过作为数据交换平台的互联网，可获得大量有关任何主题的数据。API 让程序员无须直接连接到数据库，就可以请求各种格式的数据并使用它们。这样的 API 包括以下几个。

- 纽约市交通局：通过网络提供火车、汽车和地铁交通信息，以及自动扶梯状态信息。
- 英国广播公司：提供电视和广播节目播放时间表、分类细节和图片。
- Github：提供有关 github.com 中各种数据的信息，包括用户、组织、仓库、提交和问题。
- Dark Sky：这是一个天气预报服务，通常比其他天气预报服务更准确。

应用程序开发人员已使用很多这样的 API 来开发新颖而有趣的产品和服务。例如，有很多用于 Android 系统的 Dark Sky 客户端。还有一种新的商业模式：数据提供商提供数据，客户可随心所欲地使用这些数据。

20.3 在 Go 语言中使用 JSON

Go 语言非常适合用来创建收发 JSON 的客户端和服务器。标准库提供了 encoding/json 包，可用于编码和解码 JSON 数据。

编码意味着将数据转换为编码后的格式，就本章而言，这是 JSON 格式。encoding/json 包提供了函数 Marshal，可用于将 Go 数据编码为 JSON。程序清单 20.2 显示了一个包含一些数据的 Go 语言结构体。第 7 章介绍了如何定义结构体，并指出了它们是一种绝佳的数据封装方式。

程序清单 20.2 创建结构体

```
 1:  package main
 2:
 3:  import "fmt"
 4:
 5:  type Person struct {
 6:      Name    string
 7:      Age     int
 8:      Hobbies []string
 9:  }
10:
11:  func main() {
12:      hobbies := []string{"Cycling", "Cheese", "Techno"}
13:      p := Person{
14:          Name:     "George",
15:          Age:      40,
16:          Hobbies:  hobbies,
17:      }
18:      fmt.Printf("%+v\n", p)
19:  {
```

TRY IT YOURSELF

创建要编码为 JSON 格式的结构体

在这个示例中，您将创建一个要编码为 JSON 格式的结构体。

1. 打开本书代码示例中的 hour20/example01.go。
2. 阅读其中的代码，尝试理解它是做什么的。
3. 在终端中执行命令 go run example01.go.
4. 您将在终端中看到被打印出来的结构体。

```
{Name:George Age:40 Hobbies:[Cycling Cheese Techno]}
```

▲

如果您运行这个示例，将发现其中的结构体已初始化，其中包含一些数据。要将这些数据编码为 JSON 格式，可使用函数 Marshal。这个函数接受一个接口，并返回一个字节串。由于结构体可能实现了接口，因此可将其作为参数直接传递给函数 Marshal。

```
jsonByteData, err = json.Marshall(p)
```

确定没有错误后，就可将生成的字节切片转换为字符串，并将其打印出来。

```
jsonStringData := string(jsonByteData)
fmt.Println(jsonStringData)
```

程序清单 20.3 是一个将结构体转换为 JSON 格式的完整示例。

程序清单 20.3　将结构体编码为 JSON 格式

```
 1:  package main
 2:
 3:  import (
 4:      "encoding/json"
 5:      "fmt"
 6:      "log"
 7:  )
 8:
 9:  type Person struct {
10:      Name    string
11:      Age     int
12:      Hobbies []string
13:  }
14:
15:  func main() {
16:      hobbies := []string{"Cycling", "Cheese", "Techno"}
17:      p := Person{
18:          Name:     "George",
19:          Age:      40,
20:          Hobbies:  hobbies,
21:      }
22:      fmt.Printf("%+v\n", p)
23:      jsonByteData, err := json.Marshal(p)
24:      if err != nil {
25:          log.Fatal(err)
```

```
26:         }
27:         jsonStringData := string(jsonByteData)
28:         fmt.Println(jsonStringData)
29:   }
```

TRY IT YOURSELF ▼

编码为 JSON 格式

在这个示例中，您将学习如何编码为 JSON 格式。

1. 打开本书代码示例中的 hour20/example02.go。
2. 阅读其中的代码，尝试理解它是做什么的。
3. 在终端中执行命令 go run example02.go.
4. 您将在终端中首先看到被打印出来的结构体，再看到这个结构体的 JSON 版本。

```
{Name:George Age:40 Hobbies:[Cycling Cheese Techno]}
{"Name":"George","Age":40,"Hobbies":["Cycling","Cheese","Techno"]}
```

▲

虽然前面的示例成功地将数据转换成了 JSON 格式，但存在一个问题：在 JSON 数据中，所有键名都以大写字母打头。虽然 JSON 没有官方标准，但约定是使用骆驼拼写法。表 20.1 指出 Go 变量名和 JSON 要求的变量名之间的差别。

表 20.1　　Go 和 JSON 键名

Go	JSON
Name	name
Age	age
Hobbies	hobbies

您可能认为这个问题无伤大雅，但很多 JavaScript 库都假定数据键采用的是骆驼拼写法。提供数据时，如果不按这样的约定做，就可能给使用数据的开发人员带来麻烦。乍一看，对所有的键进行分析，并让它们以小写字母开头好像是一项艰巨的任务，所幸 Go 语言的设计者已经解决了这种问题。

对于结构体，您可给其数据字段指定标签，对于带 JSON 标签的数据，将使用标签中的数据替换它。要正确地转换为 JSON 要求的骆驼拼写法，只需给结构体的字段加上标签即可。

```
type Person struct {
    Name    string   `json:"name"`
    Age     int      `json:"age"`
    Hobbies []string `json:"hobbies"`
}
```

程序清单 20.4 是一个完整的示例，演示了如何使用结构体标签来创建 JSON 要求的键。

程序清单 20.4　使用结构体标签

```
1:  package main
2:
```

```
 3:  import (
 4:      "encoding/json"
 5:      "fmt"
 6:      "log"
 7:  )
 8:
 9:  type Person struct {
10:      Name    string   `json:"name"`
11:      Age     int      `json:"age"`
12:      Hobbies []string `json:"hobbies"`
13:  }
14:
15:  func main() {
16:      hobbies := []string{"Cycling", "Cheese", "Techno"}
17:      p := Person{
18:          Name:    "George",
19:          Age:     40,
20:          Hobbies: hobbies,
21:      }
22:      fmt.Printf("%+v\n", p)
23:      jsonByteData, err := json.Marshal(p)
24:      if err != nil {
25:          log.Fatal(err)
26:      }
27:      jsonStringData := string(jsonByteData)
28:      fmt.Println(jsonStringData)
29:  }
```

如果您运行程序清单 20.4，将按要求的那样对 JSON 键进行编码。

```
go run example03.go
{Name:George Age:40 Hobbies:[Cycling Cheese Techno]}
{"name":"George","age":40,"hobbies":["Cycling","Cheese","Techno"]}
```

结构体标签还可用于指定在编码为 JSON 时忽略结构体中的空字段。默认情况下，如果结构体的字段被设置为空值，则编码为 JSON 格式后，将包含 Go 语言零值规则指定的值。

```
p := Person{}
{"name":"","age":0,"hobbies":null}
```

要指定在编码为 JSON 格式时忽略零值，可使用结构体标签指出字段可能为空，如果确实为空就忽略它。为此，可在 JSON 键名后面加上 omitempty。

```
type Person struct {
    Name    string   `json:"name,omitempty"`
    Age     int      `json:"age,omitempty"`
    Hobbies []string `json:"hobbies,omitempty"`
}
```

程序清单 20.5 的示例将一个空结构体编码为 JSON 格式。

程序清单 20.5　忽略空的结构体字段

```
 1:  package main
 2:
 3:  import (
 4:      "encoding/json"
 5:      "fmt"
 6:      "log"
 7:  )
 8:
```

```
 9:  type Person struct {
10:      Name     string    `json:"name,omitempty"`
11:      Age      int       `json:"age,omitempty"`
12:      Hobbies  []string  `json:"hobbies,omitempty"`
13:  }
14:
15:  func main() {
16:      hobbies := []string{"Cycling", "Cheese", "Techno"}
17:      p := Person{
18:          Name:     "George",
19:          Age:      40,
20:          Hobbies:  hobbies,
21:      }
22:      fmt.Printf("%+v\n", p)
23:      jsonByteData, err := json.Marshal(p)
24:      if err != nil {
25:          log.Fatal(err)
26:      }
27:      jsonStringData := string(jsonByteData)
28:      fmt.Println(jsonStringData)
29:  }
```

TRY IT YOURSELF ▼

忽略空的结构体字段

在这个示例中，您将学习如何忽略空的结构体字段。

1. 打开本书代码示例中的 hour20/example04.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中执行命令 go run example04.go。
4. 您将在终端看到打印出来的空 JSON 对象。

```
{}
```

▲

20.4 解码 JSON

JSON 解码也是一种常见的网络编程任务。收到的数据可能来自数据库、API 调用或配置文件。原始 JSON 就是文本格式的数据，在 Go 语言中可表示为字符串。函数 Unmarshal 接受一个字节切片以及一个指定要将数据解码为何种格式的接口。根据数据是如何收到的，它可能是字节切片，也可能不是。如果不是字节切片，就必须先进行转换，再将其传递给函数 Unmarshal。

```
var jsonStringData := `{"name":"George","age":40,"hobbies":["Cycling","Cheese",
"Techno"]}
jsonByteData := []byte(jsonStringData)
```

与将数据编码为 JSON 格式一样，必须定义一个接口，以指定要将数据解码为何种格式。与将数据编码为 JSON 格式一样，可使用结构体标签来告诉解码器如何将键映射到字段。

```
type Person struct {
    Name    string   `json:"name"`
    Age     int      `json:"age"`
    Hobbies []string `json:"hobbies"`
}
```

程序清单 20.6 演示了如何将 JSON 字符串数据转换为字节切片，再使用 json.Unmarshal 进行解码。

程序清单 20.6 将 JSON 数据解码为结构体

```
 1:  package main
 2:
 3:  import (
 4:      "encoding/json"
 5:      "fmt"
 6:      "log"
 7:  )
 8:
 9:  type Person struct {
10:      Name    string   `json:"name"`
11:      Age     int      `json:"age"`
12:      Hobbies []string `json:"hobbies"`
13:  }
14:
15:  func main() {
16:      jsonStringData := `{"name":"George","age":40,"hobbies":["Cycling",
    "Cheese","Techno"]}`
17:      jsonByteData := []byte(jsonStringData)
18:      p := Person{}
19:      err := json.Unmarshal(jsonByteData, &p)
20:      if err != nil {
21:          log.Fatal(err)
22:      }
23:      fmt.Printf("%+v\n", p)
24:  {
```

如果您运行程序清单 20.6，将发现 JSON 字符串被成功地转换为 Person 结构体实例了。

```
go run example05.go
{Name:George Age:40 Hobbies:[Cycling Cheese Techno]}
```

▼ TRY IT YOURSELF

解码 JSON 字符串

在这个示例中，您将学习如何解码 JSON 字符串。

1. 打开本书代码示例中的 hour20/example05.go。
2. 阅读其中的代码，尝试理解它们是做什么的。
3. 在终端中执行命令 go run example05.go。
4. 您将看到 JSON 字符串被解码为结构体。

```
{Name:George Age:40 Hobbies:[Cycling Cheese Techno]}
```

▲

20.5 映射数据类型

编码和解码 JSON 时，必须考虑 Go 和 JavaScript 表示数据类型的方式，这很重要。第 2 章说过，Go 是一种强类型语言，而 JavaScript 是一种弱类型语言，即不显式地声明变量的数据类型。下面比较了 Go 和 JavaScript 中声明字符串和整型变量的方式。

```
// JavaScript
var i = 4;
var s = "string";

// Go
var i int = 4
var s string = "string"
```

这里注意到 Go 显式地声明了变量的数据类型，而 JavaScript 没有。由于 JSON 是一个 JavaScript 子集，因此它采用的方法与 JavaScript 相同：无须声明数据类型。在强类型语言和弱类型语言之间转换数据时，这将导致一些棘手的问题。在 JSON 中，可使用的数据类型如下。

- Boolean。
- Number。
- String。
- Array。
- Object。
- Null。

程序清单 20.7 列出了一些 JSON 示例，其中涉及前述所有的数据类型。

程序清单 20.7 JSON 数据类型

```
 1:  {
 2:      "exampleBoolean":  false,
 3:      "exampleNumber":  4,
 4:      "exampleString":  "string",
 5:      "exampleArray":  ["one", "two", "three"],
 6:      "exampleObject":  {
 7:          "foo":  "bar"
 8:      },
 9:      "exampleNull":  null
10:  }
```

遗憾的是，这些数据类型不会自动映射到 Go 语言中的数据类型，因此 encoding/json 包执行显式的数据类型转换。表 20.2 显示了 JSON 数据类型和 Go 数据类型之间的对应关系。

表 20.2 Go 和 JSON 数据类型

JSON	Go
Boolean	bool
Number	float64
String	string
Array	[ ]interface{}
Object	map[string]interface{}
Null	nil

创建用于编码和解码 JSON 的结构体时，必须对上述数据类型的对应关系做到心中有数，因为如果数据类型不匹配，encoding/ json 包将引发错误。程序清单 20.8 是一个将 JSON 字符串解码为结构体的示例，您认为结果将如何呢？

程序清单 20.8 在 JSON 和 Go 之间映射数据类型

```
 1:  package main
 2:
 3:  import (
 4:      "encoding/json"
 5:      "fmt"
 6:      "log"
 7:  )
 8:
 9:  type Switch struct {
10:      On bool `json:"on"`
11:  }
12:
13:  func main() {
14:      jsonStringData := `{"on":"true"}`
15:      jsonByteData := []byte(jsonStringData)
16:      s := Switch{}
17:      err := json.Unmarshal(jsonByteData, &s)
18:      if err != nil {
19:          log.Fatal(err)
20:      }
21:      fmt.Printf("%+v\n", s)
22:  }
```

如果您运行这个示例，将出现错误，因为在 JSON 中，值 true 实际上是一个字符串，因为它被放在引号内。Go 解码器试图将这个值转换为 Go 布尔值，但由于这是一个字符串，这种转换是不可能的，因此进而引发致命错误。

```
go run example06.go
2017/08/27 17:07:13 json:  cannot unmarshal string into Go struct field Switch.on of
type bool
exit status 1
```

20.6 处理通过 HTTP 收到的 JSON

在 Go 语言中，通过 HTTP 请求获取 JSON 时，收到的数据为流而不是字符串或字节切片。在这种情况下，应使用 encoding/json 包中的另一个方法。在这个示例中，将使用 Github API。这是一个很好的 API，因为它提供格式良好的 JSON，对于有些端点，还无须验证身份。第 19 章介绍了如何创建 HTTP 客户端以通过 HTTP 协议获取数据，这种做法可直接用来从 JSON API 获取数据。

```
res, err := http.Get("https://api.github.com/users/shapeshed")
if err != nil {
log.Fatal(err)
}
defer res.Body.Close()
```

由于获取的数据为流，因此可使用 encoding/json 包中的函数 NewDecoder。这个函数接受一个 io.Reader（这正是 http.Get 返回的类型），并返回一个 Decoder。通过对返回的 Decoder

调用方法 Decode，可将数据解码为结构体。与以前一样，Decode 也接受一个结构体，因此必须创建一个结构体实例，并将其作为参数传递给 Decode。程序清单 20.9 是一个完整的示例，它调用 Github API，再将获取的数据解码为一个 Go 结构体。与以前一样，必要时可使用结构体标签将 JSON 响应中的字段映射到结构体字段。

程序清单 20.9　通过 HTTP 获取 JSON

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
)

type User struct {
    Name string `json:"name"`
    Blog string `json:"blog"`
}

func main() {
    var u User
    res, err := http.Get("https://api.github.com/users/shapeshed")
    if err != nil {
        log.Fatal(err)
    }
    defer res.Body.Close()
    err = json.NewDecoder(res.Body).Decode(&u)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%+v\n", u)
}
```

20.7　小结

本章介绍了如何在 Go 语言中处理 JSON。您学习了如何将结构体编码为 JSON 格式，以及如何使用结构体标签来支持自定义 JSON 键和可选的空值；接下来，您学习了如何解码 JSON，以及 JavaScript 和 Go 数据类型的差别；最后，您学习了如何使用 HTTP 从 API 那里获取 JSON。

20.8　问与答

问：为编码和解码 JSON，必须创建结构体，还是会自动生成结构体？

答：是的，要编码或解码 JSON，必须创建结构体。虽然这好像很繁琐，在 JSON 对象很大时尤其如此，但正如您在前面的 Github 示例中看到的，这样做可让代码更健壮、容错能力更强。如果您能将 JSON 类型映射到 Go 类型，就能将 JSON 用作数据交换格式，还可获得类型安全这样的好处。网上有多个服务可根据 JSON 数据自动创建 Go 结构。

问：为何所有的人都选择使用 JSON，而不是其他数据传输格式（如 XML）？

答：JSON 是一种灵活、易学且富有表达力的数据格式。作为 Web 语言的 JavaScript 的崛起意味着在浏览器中使用 JSON 服务的确是很容易的。另外，JSON 比其他格式占用的空间更少，这意味着可高效地传输和存储这种数据。

问：encoding/json 包提供了验证数据有效性的途径吗？

答：没有，encoding/json 没有提供验证功能。您可为结构体编写用来验证数据的方法，但编码器没有这样的方法。有多个第三方结构体验证库。

20.9 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

20.9.1 小测验

1．JavaScript 中的数字对应于哪种 Go 类型？

2．必须将 JSON 对象中的所有字段都解码到结构体中吗？

3．如果一个结构体字段可能为空，那么该使用哪个结构体标签？在这种情况下，如果该字段确实为空，结果将如何呢？

20.9.2 答案

1．JavaScript 中的数字对应于 Go 数据类型 float64。

2．不是这样的，可定义只包含您感兴趣的字段的结构体。您可使用结构体标签来将 JSON 字段映射到 Go 结构体字段。

3．如果一个字段可能为空，应给它添加结构体标签 omitempty。这样解码时，如果该字段确实为空，将忽略它。

20.10 练习

查找一个您感兴趣的服务 API，并创建一个 HTTP 客户端来使用它提供的数据。如果必要，请参阅第 19 章。获取一些 JSON 数据，并将其解码为结构体。一些有趣的 API 包括 Dark Sky、Reddit 和 Github。

第 21 章

处理文件

本章介绍如下内容。

- 文件的重要性。
- 使用 ioutil 包读写文件。
- 写入文件。
- 列出目录的内容。
- 复制文件。
- 删除文件。
- 使用文件来管理配置。

本章介绍如何使用 Go 语言来处理文件。您将学习如何读取、创建和删除文件，还有如何列出目录的内容；您还将学习使用 ioutil 包中的便利函数和使用 os 包的差别；最后，您将学习如何使用文件来管理配置。

21.1 文件的重要性

文件不过是硬盘中的数据，看起来好像没什么了不起，但实际上，文件能够让程序员管理配置、存储程序的状态乃至从底层操作系统中读取数据。

UNIX 型操作系统（Linux 和 macOS）的一个重要特征是，将一切都视为文件。这意味着在操作系统看来，从键盘到打印机的所有东西都可像文件那样编址。在这方面，UNIX 走得更远，它通过虚拟文件系统来暴露系统信息。这意味着可像读取文件一样读取系统数据。

在 UNIX 系统中，可使用命令 cat 来读取文件的内容并将其打印到终端。鉴于 UNIX 系统以文件的方式暴露系统数据，因此命令 cat 也可用来提取有关底层系统的信息。/proc/loadavg 就是这样的一个虚拟文件，它包含有关系统当前负载的信息。

```
cat /proc/loadavg
0.31 0.25 0.26 1/227 15992
```

如果您再次执行这个命令，将发现显示的值变了，这表明有关系统负载的数据是实时的。

```
cat /proc/loadavg
0.47 0.30 0.28 2/229 16571
```

在大多数 UNIX 型系统中，可使用命令 watch 来创建有关系统负载的实时视图，这种视图每隔 2s 更新一次，即每隔 2s 读取文件/proc/loadavg 中的数据，并将其显示到屏幕上。

```
watch cat /proc/loadavg
Every 2.0s:  cat /proc/loadavg
0.32 0.30 0.28 1/232 18308
```

只需一行代码，就创建了一个有关系统负载的实时视图！这都是拜 UNIX 将一切视为文件所赐。文件并非总是无声的数据，通过读取文件，可创建根据操作系统的状态做出反应的程序。

21.2 使用 ioutil 包读写文件

鉴于处理文件是一种很常见的任务，标准库提供了 ioutil 包，让您能够快速执行众多涉及读写文件的操作。实际上，这个包几乎就是一个 os 模块包装器，使用它需要编写的代码更少，且无须执行清理操作。如果您需要执行下述任何操作，且无须做细致的控制，则使用 ioutil 包是不错的选择。

- 读取文件。
- 列出目录的内容。
- 创建临时目录。
- 创建临时文件。
- 创建文件。
- 写入文件。

21.2.1 读取文件

读取文件是最常见的操作之一。ioutil 包提供了函数 Readfile，您可使用它来完成这项任务，这个函数将一个文件名作为参数，并以字节切片的方式返回文件的内容。这意味着如果要将文件内容作为字符串使用，则必须将返回的字节切片转换为字符串。程序清单 21.1 读取一个文件并将其内容打印到终端。

程序清单 21.1 读取一个文件并将其内容打印到终端

```
 1:  package main
 2:
 3:  import (
```

```
 4:      "fmt"
 5:      "io/ioutil"
 6:      "log"
 7:  )
 8:
 9:  func main() {
10:      fileBytes, err := ioutil.ReadFile("example01.txt")
11:      if err != nil {
12:          log.Fatal(err)
13:      }
14:
15:      fmt.Println(fileBytes)
16:
17:      fileString := string(fileBytes)
18:      fmt.Println(fileString)
19:  }
```

对程序清单 21.1 解读如下。

- 使用 ioutil 包中的函数 Readfile 读取文件。
- 这个函数返回一个字节切片。
- 将返回的字节切片转换为字符串。
- 将字符串打印到终端，以显示文件的内容。

TRY IT YOURSELF ▼

读取文件

在这个示例中，您将明白在 Go 语言中如何读取文件。

1. 打开本书代码示例中的 hour21/example01.go。
2. 在终端中执行命令 go run example01.go。
3. 您将在终端中看到被打印出来的文件内容。

▲

21.2.2 创建文件

ioutil 包还提供了用于创建文件的便利函数 WriteFile。这个函数设计用于将数据写入文件，但也可使用它来创建文件。函数 WriteFile 接受一个文件名、要写入文件的数据以及应用于文件的权限。

文件权限是从 UNIX 权限衍生而来的，它们对应于 3 类用户：文件所有者、与文件位于同一组的用户、其他用户。处理文件时，理解文件权限是确保安全的重要方面，因为错误地设置权限意味着数据可能被原本不应该让他读取的人获得。

Go 语言使用 UNIX 权限的数字表示法，很多用于处理文件的函数都将权限值作为参数。表 21.1 说明了数字表示法和符号表示法。

表 21.1 文件权限

符号表示法	数字表示法	说明
----------	0000	无权限
-rwx------	0700	只有所有者能够读取、写入和执行
-rwxrwx---	0770	所有者及其所在的用户组能够读取、写入和执行
-rwxrwxrwx	0777	所有人都能够读取、写入和执行
---x--x--x	0111	所有人都能够执行
--w--w--w-	0222	所有人都能够写入
--wx-wx-wx	0333	所有人都能够写入和执行
-r--r--r--	0444	所有人都能够读取
-r-xr-xr-x	0555	所有人都能够读取和执行
-rw-rw-rw-	0666	所有人都能够读取和写入
-rwxr-----	0740	所有者能够读取、写入和执行，而所有者所在的用户组能够读取

符号表示法是数字表示法的视觉表示。符号表示法总共包含 10 个字符。最左边的字符指出了文件是普通文件、目录还是其他东西，如果这个字符为-，就表示文件为普通文件；接下来的 3 个字符指定了文件所有者的权限；再接下来的 3 个字符表示所有者所在用户组的权限；而最后 3 个字符表示其他人的权限。

从表 2.11 可知，数字 0777 表示所有人都有全部的权限，而相应的符号表示法没有缺失任何字符；相反，数字权限 0700 表示只有所有者才能对文件执行任何操作。

在 UNIX 型系统中，文件的默认权限为 0644，即所有者能够读取和写入，而其他人只能读取。在文件系统中创建文件时，应考虑如何给它指定权限。如果不确定该如何指定权限，使用默认的 UNIX 权限就可以了。

程序清单 21.2 在文件系统中创建了一个文件，并将其权限设置为 0644。

程序清单 21.2 在文件系统中创建了一个文件，并将其权限设置为 0644

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "io/ioutil"
 6:      "log"
 7:  )
 8:
 9:  func main() {
10:      b := make([]byte, 0)
11:      err := ioutil.WriteFile("example02.txt", b, 0644)
12:      if err != nil {
13:          log.Fatal(err)
14:      }
15:  }
```

对程序清单 21.2 解读如下。

- 函数 WriteFile 接受一个字节切片，因此创建一个空字节切片，并将其赋给变量 b。

- 调用函数 WriteFile，并向它传递文件名、空字节切片以及要给文件设置的权限。
- 如果没有错误，将创建指定的文件。

这里给函数 WriteFile 传递了空字节切片，这是一种使用 ioutil 包中便利函数的技巧。函数 WriteFile 在指定的文件不存在时创建它，因此也可使用这个函数来创建空文件。

如果您运行这个示例，将创建一个空文件，并给它设置指定的权限。

```
go run example02.go
ls -l
-rw-r--r--   1 go go 172 Aug 29 09:44 example02.go
-rw-r--r--   1 go go   0 Sep  2 12:32 example02.txt
```

TRY IT YOURSELF ▼

创建文件

在这个示例中，您将明白如何在 Go 语言中创建文件。

1. 打开本书代码示例中的 hour21/example02.go。
2. 在终端中执行命令 go run example02.go。
3. 将创建新文件 example02.txt。

▲

21.3 写入文件

正如您预期的，函数 WriteFile 也可用来写入文件。程序清单 21.2 演示了如何创建空文件，这是通过传入一个空字节切片实现的。要写入文件，只需传入一些值，而不是传入空字节切片。要将字符串写入文件，必须先将其转换为字节切片。程序清单 21.3 演示了如何将文本字符串写入文件。如果指定的文件不存在，将创建它。

程序清单 21.3 将文本字符串写入文件

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "io/ioutil"
 6:      "log"
 7:  )
 8:
 9:  func main() {
10:      s := "Hello World"
11:      err := ioutil.WriteFile("example03.txt", []byte(s), 0644)
12:      if err != nil {
13:          log.Fatal(err)
14:      }
15:
```

▼TRY IT YOURSELF

写入文件

在这个示例中，您将明白如何写入文件。

1. 打开本书代码示例中的 hour21/example03.go。
2. 在终端中执行命令 go run example03.go。
3. 这将创建一个新文件——example03.text。
4. 请打开这个文件。您将看到在其中写入了一些文本。

▲

21.4 列出目录的内容

要处理文件系统中的文件，必须知道目录结构。为此，ioutil 包提供了便利函数 ReadDir，它接受以字符串方式指定的目录名，并返回一个列表，其中包含按文件名排序的文件。文件名的类型为 FileInfo，包含如下信息。

- Name：文件的名称。
- Size：文件的长度，单位为字节。
- Mode：用二进制位表示的权限。
- ModTime：文件最后一个被修改的时间。
- IsDir：文件是否是目录。
- Sys：底层数据源。

程序清单 21.4 列出了一个目录中的文件及其权限。

程序清单 21.4 列出目录中的文件及其权限

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "io/ioutil"
 6:      "log"
 7:  )
 8:
 9:  func main() {
10:      files, err := ioutil.ReadDir(".")
11:      if err != nil {
12:          log.Fatal(err)
13:      }
14:
15:      for _, file := range files {
16:          fmt.Println(file.Mode(), file.Name())
17:      }
18:  }
```

TRY IT YOURSELF

列出目录的内容

在这个示例中，您将明白如何列出目录的内容。

1. 打开本书代码示例中的 hour21/example04.go。
2. 在终端中执行命令 go run example04.go。
3. 您将在终端中看到目录的内容。

21.5 复制文件

ioutil 包可用于执行一些常见的文件处理操作，但要执行更复杂的操作，应使用 os 包。os 包运行在稍低的层级，因此使用它时，必须手工关闭打开的文件。如果您阅读 os 包的源代码，将发现 ioutil 包中的很多函数都是 os 包包装器，您无须显式地关闭文件。

要复制文件，只需结合使用 os 包中的几个函数。以编程方式复制文件的步骤如下。

1. 打开要复制的文件。
2. 读取其内容。
3. 创建并打开要将这些内容复制到其中的文件。
4. 将内容写入这个文件。
5. 关闭所有已打开的文件。

程序清单 21.5 是一个完整的示例，演示了如何读取一个既有文件的内容，并将其复制到一个新文件中。

程序清单 21.5 将文件的内容复制到一个新文件中

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "log"
 6:      "os"
 7:  )
 8:
 9:  func main() {
10:      from, err := os.Open("./example05.txt")
11:      if err != nil {
12:          log.Fatal(err)
13:      }
14:      defer from.Close()
15:
16:      to, err := os.OpenFile("./example05.copy.txt", os.O_RDWR|os.O_CREATE, 0666)
17:      if err != nil {
18:          log.Fatal(err)
19:      }
```

```
20:       defer to.Close()
21:
22:       _, err = io.Copy(to, from)
23:       if err != nil {
24:           log.Fatal(err)
25:       }
26:   }
```

对程序清单 21.5 解读如下。

- 使用 os 包中的函数 Open 来读取磁盘文件。
- 使用 defer 语句在程序完成其他所有操作后关闭文件。
- 使用函数 OpenFile 打开文件。第一个参数是要打开（如果不存在，就创建）的文件的名称；第二个参数是用于文件的标志，在这里指定的是读写文件，并在文件不存在时创建它；最后一个参数设置文件的权限。
- 再次使用 defer 语句在执行完其他操作后关闭文件。
- 使用 io 包中的函数 Copy 复制源文件的内容，并将其写入目标文件。

▼ TRY IT YOURSELF

复制文件

在这个示例中，您将明白如何复制文件。

1. 打开本书代码示例中的 hour21/example05/example05.go。
2. 在终端中执行命令 go run example05.go。
3. 这将创建一个新文件，其内容与 example05.txt 相同。

▲

21.6 删除文件

在程序员看来，删除文件通常是件坏事。文件删除后，就再也找不回来了，代码中删除文件的 Bug 可能导致灾难，在文件没有备份时尤其如此。良好的做法是尽可能对数据进行“软删除”，这样，如果代码有 Bug 或者以后改变了注意，就可以恢复删除的数据。然而，在有些情况下，必须将文件或文件夹删除，os 包提供了函数 Remove，它能够让您轻松地完成这种任务。需要指出的是，使用这个函数时，不会发出警告，您也无法将删除的文件恢复，因此务必要谨慎。

程序清单 21.6 演示了如何删除文件。

程序清单 21.6 删除文件

```
1:  package main
2:
3:  import (
4:      "log"
```

```
 5:      "os"
 6:  )
 7:
 8:  func main() {
 9:      err := os.Remove("./deleteme.txt")
10:      if err != nil {
11:          log.Fatal(err)
12:      }
13:  }
```

TRY IT YOURSELF ▼

删除文件

在这个示例中，您将明白如何删除文件。

1. 打开本书代码示例中的 hour21/example06/example06.go。
2. 在这个文件夹中，有一个名为 deleteme.txt 的文件。
3. 在终端中执行命令 go run example06.go。
4. 核实文件 deleteme.txt 已被删除。

▲

21.7 使用文件来管理配置

在编程中常使用文件来管理配置。考虑到代码可能在不同的环境中执行，可使用一个文件来设置各种用于启动程序的配置参数。在开发过程中，应用程序将从开发环境移到生成环境中，而使用文件是管理环境差异的有效方式。这种环境差别包括如下几个方面。

- Web 服务的 URL。
- 访问密钥。
- 端口号。
- 环境变量。

21.7.1 使用 JSON 文件

第 20 章介绍了如何处理 JSON。在声明可存储在文件中并在必要时读取的配置方面，JSON 是一种卓有成效的标准方式。将配置存储在文件中的另一个优点是，可对其进行版本控制并集成到自动构建过程中。第 20 章说过，JSON 是一种声明键值的简单方式，您可将这些键值解码为 Go 结构体，再使用它们。

```
{
  "name":  "George",
  "awake":  true,
  "hungry":  false
}
```

通过将配置存储在 JSON 文件中，Go 程序可读取这个文件，并将其作为配置数据使用。程序清单 21.7 读取了一个 JSON 文件，并将其解码为一个配置结构体。

程序清单 21.7 使用 JSON 存储配置

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
)

type Config struct {
    Name   string `json:"name"`
    Awake  bool   `json:"awake"`
    Hungry bool   `json:"hungry"`
}

func main() {
    f, err := ioutil.ReadFile("config.json")
    if err != nil {
        log.Fatal(err)
    }
    c := Config{}
    err = json.Unmarshal(f, &c)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%+v\n", c)
}
```

这个示例使用了本章前面介绍的函数 ReadFile，并像第 20 章介绍的那样将文件的内容解码为一个结构体。只需使用标准库编写几行代码，就可根据运行环境来配置程序。如果需要更多配置，可扩展 Go 结构体，并在 JSON 对象中添加相应的字段。

▼ TRY IT YOURSELF

从 JSON 文件中读取配置

在这个示例中，您将明白如何从 JSON 文件中读取配置。

1. 打开本书代码示例中的 hour21/example07/example07.go。
2. 在这个文件夹中，有一个 JSON 文件，其中包含一些配置。
3. 在终端中执行命令 go run example07.go。
4. 这将读取 JSON 文件中的配置，并将其打印到控制台。

▲

21.7.2 使用 TOML 文件

TOML（Tom's Obvious, Minimal Language）是一种专为存储配置文件而设计的格式，它

在 Go 社区中深受欢迎。相比于 JSON，其表达能力更强，且更容易映射 Go 类型。JSON 是为序列化数据而设计的，而 TOML 是专门为存储配置文件而设计的，因此 TOML 相比于 JSON 有一定的优势，如更容易阅读、具备 JSON 没有的特性（如注释）。在基本层面，TOML 的语法非常简单，可像 JSON 那样指定键和值。

```
Name = "George"
Awake = true
Hungry = false
```

与 JSON 一样，TOML 也不是 Go 的组成部分，它可用于任何语言。Go 标准库中没有支持 TOML 的包，虽然您可编写一些代码来分析 TOML 文件，但有一些卓越的第三方 TOML 包也可以使用。编写本书期间，最流行的 TOML 包之一是 BurntSushi 编写的。这个包能够让您轻松地在 Go 语言中将 TOML 用作配置文件格式，但由于它并不包含在标准库中，因此您必须单独安装。

```
go get github.com/BurntSushi/toml
```

这个包提供了函数 DecodeFile，这个函数接受一个文件名以及一个要将 TOML 解码到其中的结构体。程序清单 21.8 演示了如何将 TOML 文件解码为配置结构体。

程序清单 21.8　使用 TOML 文件来存储配置

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "log"
 6:
 7:      "github.com/BurntSushi/toml"
 8:  )
 9:
10:  type Config struct {
11:      Name string
12:      Awake bool
13:      Hungry bool
14:  }
15:
16:  func main() {
17:      c := Config{}
18:      _, err := toml.DecodeFile("config.toml", &c)
19:      if err != nil {
20:          log.Fatal(err)
21:      }
22:      fmt.Printf("%+v\n", c)
23:  }
```

相比于前面的 JSON 示例，这里从文件中读取配置的代码更少，这是因为 TOML 是一种表达力更强、更容易理解的配置格式，也正是因为如此，这里介绍的 TOML 包才如此深受欢迎。

TRY IT YOURSELF ▼

从 TOML 文件中读取配置

在这个示例中，您将明白如何从 TOML 文件中读取配置。

1. 打开本书代码示例中的 hour21/example08/example08.go。
2. 在这个文件夹中，有一个 TOML 文件，其中包含一些配置。
3. 在终端中执行命令 go run example08.go。
4. 将读取 TOML 文件中的配置，并将其打印到控制台。

▲

是否选择使用第三方包完全取决于个人偏好，但在决定自己编写代码还是使用第三方包时，应三思而后行。标准库中的代码与未来的 Go 版本兼容的可能性很大，且 Go 语言项目很看重稳定性。而使用第三方包时，质量和持续性是没有保证的，因为其开发者可能不再维护它，而将精力放在其他工作上。一般而言，Go 社区流行的项目都得到了很好的维护，提供的包的质量也非常高，可为您节省大量的时间。

就该使用 JSON 和标准库还是 TOML 和第三方包这个问题，没有唯一的答案。如果您喜欢 TOML 丰富的表达力，在项目中添加维护良好的依赖并非坏事，但如果使用 JSON 文件就能表示所需的配置，便可少管理一个依赖。

21.8 小结

本章介绍了如何在 Go 语言中处理文件。您了解了 ioutil 和 os 包的差别，还知道 ioutil 包提供了多个便利的文件处理方法；接下来，您学习了文件权限及其对安全的影响；最后，您学习了如何使用 JSON 和 TOML 文件来管理配置。

21.9 问与答

问：ioutil 包为何没有提供用于执行复制文件等操作的函数？

答：Go 语言致力于确保核心库为小巧而轻量级的。另外，不同的操作性系统差别很大，这导致创建通用的文件复制方法是很难的。鉴于此，没有用于复制文件的便利方法，要复制文件，应使用 os 包。

问：操作文件前，Go 语言是否会核实文件确实存在？

答：不会，程序员必须在使用文件前核实它确实存在。如果文件不存在，将引发错误。

问：如何确定该给文件设置什么样的权限？

答：如果不确定该给文件设置什么样的权限，应采取保守的态度，设置尽可能低的权限，默认的 UNIX 权限 0644 就是不错的选择。设置过于宽松的权限（如 0777）意味着如果服务器被攻破，则攻击者将能够随意处置文件，包括将其删除。

21.10 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看

后面的答案。

21.10.1 小测验

1. 如果操作系统中的文件被设置为权限 0700，Go 语言程序能够读取它吗？
2. 相比于 JSON，使用 TOML 有哪些优点？
3. 使用第三方包意味着什么？

21.10.2 答案

1. 这是一个带有陷阱的问题！Go 程序在用户的上下文中执行。权限为 0700 的文件只有所有者才能使用，如果执行 Go 程序的用户就是文件的所有者，则程序将能够读取这个文件；否则程序将不能访问它。

2. TOML 是一种专门为存储配置文件而设计的数据格式，因此支持在文件中包含注释，这有助于提高配置文件的可读性和可维护性。使用 TOML 格式时，也更容易表示 Go 数据类型。

3. 第三方包可能提供高品质而维护良好的代码，能够为您节省大量的时间。然而，使用第三方包确实会在项目中引入依赖，因此使用可能在几个月后消失的第三方包可能给您增加很大的负担。Go 社区有一些极有才华的开发人员，他们开发了一些卓越的包。只要就包的存活时间及维护情况做些调查，选择的包最终给您带来负担的可能性就会很小。

21.11 练习

假设您要编写一个小型程序，将您养的第一只宠物的信息打印出来。为描述这个宠物，可创建一个结构体，也可创建一个 JSON 或 TOML 配置文件。请探索如何在 JSON 和 TOML 配置文件中表示不同的数据类型，如数组、整数和浮点数。

第 22 章

正则表达式简介

本章介绍如下内容。

- 定义正则表达式。
- 熟悉正则表达式语法。
- 将正则表达式用于数据验证。
- 将正则表达式用于数据变换。

大多数语言都支持神秘而神奇的正则表达式，它们提供了一种强大的模式匹配和数据分析方式。乍一看，正则表达式不好理解，但花点时间来学习它们是值得的，因为它们表达力丰富且功能强大。掌握正则表达式后，您将发现它们是编程武器库中的利器。本章介绍正则表达式以及在 Go 语言中使用正则表达式的一些方式。

22.1 定义正则表达式

正则表达式描述了可用于与数据交互的搜索模式。正则表达式虽然难学，但其功能非常强大。使用正则表达式可完成验证数据、查找数据以及操作大量文本等任务。相比于其他方法，表达式查找和模式匹配的效率要高得多，但正则表达式学习起来的确很难。正则表达式犹如一门微型编程语言，要掌握它们需要花些时间。然而，即便是初学者，也完全能够使用正则表达式。

正则表达式的用途之一是在字符串中查找与指定正则表达式匹配的子串，这常被称为大海捞针。比如，在句子 Chocolate is my favorite!中查找单词 chocolate。您可能认为这个句子包含 chocolate，但正则表达式检查是区分大小写的，因此能否找到 chocolate 取决于您使用的正则表达式。

在 Go 语言中，正则表达式功能是由 regex 包提供的，这个包实现了正则表达式的查找和模式匹配功能。它使用的是 RE2 语法，这大致与 Perl 和 Python 使用的语法相同。它操作的

目标可以是字符串，也可以是字节。

要在大海里捞针，可使用函数 MatchString，它接受一个正则表达式模式和一个字符串，并根据是否匹配返回 true 或 false。程序清单 22.1 是一个查找单词 chocolate 的示例。

程序清单 22.1 大海捞针

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "log"
 6:      "regexp"
 7:  )
 8:
 9:  func main() {
10:      needle := "chocolate"
11:      haystack := "Chocolate is my favorite!"
12:      match, err := regexp.MatchString(needle, haystack)
13:      if err != nil {
14:          log.Fatal(err)
15:      }
16:      fmt.Println(match)
17:  }
```

如果您运行这个程序，它将打印 false，这表明不匹配。怎么会这样呢？因为正则表达式区分大小写，因此没有找到字符串 chocolate。

TRY IT YOURSELF ▼

查找字符串

在这个示例中，您将明白如何使用正则表达式来查找字符串。

1. 在终端中切换到文件夹 hour22/。
2. 在终端中执行命令 go run example01。
3. 您将看到使用正则表达式查找的结果为 false。

▲

要以不区分大小写的方式查找，必须修改正则表达式，使其在查找单词时不区分大小写。为指定查找时不区分大小写，需要使用一种特殊语法。

```
needle := "(?i)chocolate"
```

这个字符串开头的特殊语法让正则表达式引擎不区分大小写。如果您运行这个示例，结果将为 true，因为找到了指定的字符串。

```
go run example02.go
true
```

TRY IT YOURSELF

不区分大小写

在这个示例中，您将明白如何让正则表达式执行查找时不区分大小写。

1. 在终端中切换到文件夹 hour22/。
2. 在终端中执行命令 go run example02。
3. 您将看到使用正则表达式查找的结果为 true。

22.2 熟悉正则表达式语法

详细介绍正则表达式语法不在本书的范围内，但创建正则表达式时，您可能会使用一些常见的语法。表 22.1 列出了一些最常用的字符。

表 22.1 常用的正则表达式语法

字符	含义
.	与除换行符之前的其他任何字符都匹配
*	与零个或多个指定的字符匹配
^	表示行首
$	表示行尾
+	匹配一次或多次
?	匹配零或一次
[]	与方括号内指定的任何字符都匹配
{n}	匹配 n 次
{n,}	匹配 n 次或更多次
{m,n}	最少匹配 m 次，最多匹配 n 次

为探索正则表达式语法，这里假设将用户名插入数据库前需要先验证它。编写正则表达式前，最好先将要检查的条件写下来。在这里，应根据如下规则来匹配字符串。

- 应长于 4 个字符，但不超过 12 个字符。
- 应只包含字母和数字。
- 字符可大写，也可小写。

请尝试在不使用正则表达式的情况下编写检查这些条件的代码。这不是不可能，但需要编写大量的代码。通过使用正则表达式，只需一行就能完成这种任务。

```
^[a-zA-Z0-9]{5-12}$
```

只需这几个字符就能完成这种任务，好像很神奇。对这行内容解读如下。

- 字符 ^ 表示从字符串开头开始匹配。

- 方括号（[]）内的字符集表示与其中的任何字符都匹配。
- 大括号（{}）内的数字表示应至少匹配 5 次，但最多不超过 12 次。

22.3 使用正则表达式验证数据

正则表达式可用于验证程序的输入数据，这是一种分析和理解数据的高效方式。要将正则表达式赋给变量，必须先对其进行分析。用于分析正则表达式的函数有两个。

- Compile：在正则表达式未能通过编译时返回错误。
- MustCompile：在正则表达式无法编译时引发 panic。

该使用哪一个取决于具体情况，但 MustCompile 通常是更佳的选择。

```
re := regexp.MustCompile("^[a-zA-Z0-9]{5,12}")
```

程序清单 22.2 演示了如何使用正则表达式来检查用户名。

程序清单 22.2 使用正则表达式来验证数据

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "regexp"
 6:  )
 7:
 8:  func main() {
 9:      re := regexp.MustCompile("^[a-zA-Z0-9]{5,12}")
10:      fmt.Println(re.MatchString("slimshady99"))
11:      fmt.Println(re.MatchString("!asdf£33£3"))
12:      fmt.Println(re.MatchString("roger"))
13:      fmt.Println(re.MatchString("iamthebestuseofthisappevaaaar"))
14:  }
```

程序清单 22.2 检查每个字符串，并根据它是否与正则表达式匹配打印 true 或 false。在实际程序中，可使用正则表达式来检查用户名，确定将其插入数据库是否是安全的。

```
go run example03.go
true
false
true
true
```

TRY IT YOURSELF ▼

使用正则表达式验证数据

在这个示例中，您将明白如何使用正则表达式来验证数据。

1. 在终端中切换到文件夹 hour22/。
2. 在终端中执行命令 go run example03。
3. 将根据正则表达式对输入字符串进行评估。

▲

22.4 使用正则表达式来变换数据

在前一个示例中，有些用户名是无效的，因为其长度不正确或包含非法字符。另一个常见的编程任务是对数据进行清洗，以便能够安全地使用它们。

正则表达式也可用于完成这种任务——匹配模式并进行替换。程序清单 22.2 演示了如何使用正则表达式来检查用户名，以确认它是否符合特定的模式。可对这个程序进行改进，对不符合正则表达式的用户名进行清洗，以便依然能够使用它们。

来复习一些有关用户名的规则：包含的字符不能超过 12 个。要清洗太长的用户名，可先将其截断为只包含 12 个字符。然后根据正则表达式对其进行评估，看看它是否包含非法字符。如果包含非法字符，可将其替换为合法字符。

经常需要创建这样的小型数据管道来清洗数据，在这种情况下，正则表达式是非常便利的工具。程序清单 22.3 对一系列用户名进行评估，检查它们是否满足有关用户名的规则。如果用户名太长，就截断；如果包含非法字符，就将其替换为另一个合法字符。

程序清单 22.3 使用正则表达式来变换数据

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "regexp"
 6:  )
 7:
 8:  func main() {
 9:      usernames := [4]string{
10:          "slimshady99",
11:          "!asdf£33£3",
12:          "roger",
13:          "Iamthebestuserofthisappevaaaar",
14:      }
15:
16:      re := regexp.MustCompile("^[a-zA-Z0-9]{5,12}")
17:      an := regexp.MustCompile("[[:^alnum:]]")
18:
19:      for _, username := range usernames {
20:          if len(username) > 12 {
21:              username = username[:12]
22:              fmt.Printf("trimmed username to %v\n", username)
23:          }
24:          if !re.MatchString(username) {
25:              username = an.ReplaceAllString(username, "x")
26:              fmt.Printf("rewrote username to %v\n", username)
27:          }
28:      }
29:  }
```

如果您运行这个程序，将发现用户名 Iamthebestuserofthisappevaaaar 被截取为 Iamthebestus，而用户名!asdf?33?3 中的非法字符都被替换为 x。

```
go run example04.go
rewrote username to xasdfx33x3
trimmed username to Iamthebestus
```

TRY IT YOURSELF ▼

使用正则表达式变换数据

在这个示例中，您将明白如何使用正则表达式来变换数据。

1. 在终端中切换到文件夹 hour22/。
2. 在终端中执行命令 go run example04。
3. 您将看到对输入字符串进行了截取和转换，使其与正则表达式匹配。

▲

22.5 小结

本章介绍了正则表达式——一种功能强大的模式匹配和替换方式。您学习了如何在大海里捞针，还有一些常用的正则表达式语；接下来，您见识了两个分别使用正则表达式来验证和清洗数据的示例；最后，您学习了如何使用正则表达式来分析文本，以提取感兴趣的数据。

22.6 问与答

问：在什么地方能够更深入地学习正则表达式语法？

答：正则表达式语法学习起来很难，因此市面上有多部专门介绍正则表达式的著作，还有很多卓越的基于浏览器的正则表达式测试器。您可使用这些测试器来测试正则表达式并获得更多的正则表达式知识。

问：与使用 strings 包相比，使用正则表达式的效率更高还是更低？

答：strings 包包含几个让您能够在大海里捞针的函数，但它不支持复杂的模式匹配。一般而言，使用正则表达式的速度更慢。第 15 章介绍了性能测试，这里提供了一个很好的案例，您可使用基准测试来检查哪种代码的运行速度更快！

问：有人能记住所有的正则表达式语法吗？

答：有些人擅长创建正则表达式，能够将全部正则表达式语法都牢记在心。对普通大众来说，创建复杂正则表达式时都需参阅文档并对模式进行测试。

22.7 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

22.7.1　小测验

1．函数 Compile 和 MustCompile 之间有何不同？

2．如何让正则表达式匹配时不区分大小写？

3．哪个正则表达式字符表示行尾？

22.7.2　答案

1．当正则表达式无法分析时，函数 Compile 返回错误，而函数 MustCompile 引发 panic。

2．标志 i 让正则表达式不区分大小写。

3．字符$表示行尾。

22.8　练习

1．编写与这样的字符串匹配的正则表达式：长 1～10 个字符，且只包含数字。您可能想使用正则表达式在线生成器。

2．编写与任何只包含大写字母的字符串匹配的正则表达式。您可能想使用正则表达式在线生成器。

第 23 章

Go 语言时间编程

本章介绍如下内容。

- 时间元素编程。
- 让程序休眠。
- 设置超时时间。
- 使用 **ticker**。
- 以字符串格式表示时间。
- 使用结构体 **Time**。
- 时间加减。
- 比较两个不同的 **Time** 结构体。

本章介绍 Go 语言时间编程。您将学习如何显示当前时间以及如何使用 ticker 和超时。您将学习如何使用 Go 语言分析字符串表示的时间，以及如何访问时间中的日期和月份等。您还将学习如何比较两个时间。阅读完本章后，您将对如何在 Go 语言中处理时间有深入的认识。

23.1 时间元素编程

时间是一个重要的编程元素，可用于计算、同步服务器以及测量。Go 语言标准库提供了 time 包，其中包含用于同当前时间交互以及测量时间的函数和方法。

在编程中，时间通常被称为“实时”“过去的时间”和“壁挂钟”。对于术语“壁挂钟”，可将其视为挂在墙上的时钟。它显示的时间随时区或地区性调整而异，但每隔 24h 显示的时间都相同。对编程来说，这种随时区、地区性调整而异的二十四小时制并不理想，因此也存在单调时钟（monotonic clock）的概念。单调时钟以固定的方式度量时间，可用于可靠地度量事件发生的时间。

要使用 Go 语言打印计算机中的当前时间，可使用函数 Now。程序清单 23.1 所示的示例演示了如何获悉当前时间。

程序清单 23.1 打印当前时间

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func main() {
 9:      fmt.Println(time.Now())
10:  }
```

如果您运行这个程序，它将打印计算机的当前时间。根据计算机的地区设置，您看到的打印出来的时间格式可能不同。

```
go run example01.go
2017-09-03 13:38:04.608932763 +0100 BST
```

这个时间是怎么来的呢？难道 Go 语言有神奇的时钟可供参考吗？实际上，它来自底层操作系统。取决于底层操作系统中的时间准确度，这种时间可能很有用，也可能没有用。在大多数操作系统中，用户都可设置时间。

例如，在 Linux 系统中，您可进行时间旅行，将时间设置为未来的。下面的命令将时间设置为 2050 年元旦。

```
sudo date +%Y%m%d -s "20500101"
```

如果您再次运行程序清单 23.1，将发现这些代码运行的时间与前面完全不同。

```
go run example01.go
2050-01-01 00:01:19.186167091 +0000 GMT
```

如您所见，时间受众多变数的影响，其中包括在操作系统中设置的时间不正确。鉴于此，很多系统管理员会安装将时间与网络时钟同步的服务。网络时间协议（Network Time Protocol，NTP）是一种在整个网络中同步时间的网络协议，使用 NTP 的不同计算机更有可能就时间达成一致，但在本地它们依然可以设置不同的时区。

在计算中，要消除时区的影响，可参照世界标准时间（Coordinated Universal Time，UTC）。UTC 是时间标准而非时区，它让不同地区的计算机有相同的参照物，而不用考虑相对时区。

很多操作系统都默认使用 NTP，同时允许用户设置时间。将软件部署到服务器时，必须确定准确的时间，因此推荐您安装 NTP 服务。

▼ TRY IT YOURSELF

将时间打印到终端

在这个示例中，您将明白如何在 Go 语言中获取当前时间。

1. 在终端中切换到文件夹 hour23/。
2. 在终端中执行命令 go run example01。
3. 您将看到您的计算机的当前时间被打印到终端中。
4. 尝试在您的计算机上修改时间，然后再次运行这个示例。

23.2 让程序休眠

知道时间编程的复杂性后，该来让程序休眠了。在计算机程序中，休眠意味着暂停执行程序。在休眠期间，程序什么都不做。但在 Go 语言中，如果 Goroutine 处于休眠状态，则程序的其他部分可继续执行。程序清单 23.2 的示例让程序休眠 3s，再唤醒它。这个程序被唤醒后，将向控制台打印指出这一点的消息。

程序清单 23.2 休眠 3s

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func main() {
 9:      time.Sleep(3 * time.Second)
10:      fmt.Println("I'm awake")
11:  }
```

TRY IT YOURSELF

使用 Sleep 暂停

在这个示例中，您将明白如何让程序休眠 3s。

1. 在终端中，切换到文件夹 hour23/。
2. 在终端中执行命令 go run example02。
3. 您将看到程序将休眠 3s，并在醒来后向终端打印一条消息。

为让程序暂停执行一会儿，休眠是很有用的。可通过休眠来等待其他任务完成或让程序暂停执行。但除非只是想让程序暂停一会儿，否则使用 Goroutine 来管理执行流程是更佳的选择。

23.3 设置超时时间

第 12 章介绍了通道，您知道了可在 select 语句中指定超时时间。这是使用 time 包在过

去特定的时间后向通道发送一条消息实现的。要在特定的时间过后执行某项操作，可使用函数 After。程序清单 23.3 的示例使用函数 After 在特定时间过后触发超时。

程序清单 23.3 使用函数 After 触发超时

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func main() {
 9:      fmt.Println("You have two seconds to calculate 19 * 4")
10:      for {
11:          select {
12:          case <-time.After(2 * time.Second):
13:              fmt.Println("Time's up! The answer is 74. Did you get it?")
14:              return
15:          }
16:      }
17:  }
```

如果您运行这个示例，将显示一条消息，2s 后再打印答案并退出。

```
go run example03.go
You have two seconds to calculate 19 * 4
Time's up! The answer is 74. Did you get it?
```

▼ TRY IT YOURSELF

使用超时

在这个示例中，您将明白如何设置超时时间。

1. 在终端中切换到文件夹 hour23/。
2. 在终端中执行命令 go run example03。
3. 您将看到一条消息，让您完成一个数学题。
4. 2s 后再显示另一条消息，然后程序退出。

▲

23.4 使用 ticker

使用 ticker 可让代码每隔特定的时间就重复执行一次。需要在很长的时间内定期执行任务时，这么做很有用。

程序清单 23.4 使用了 ticker 来每隔 5s 重新显示时间。

程序清单 23.4 使用 ticker 显示时间

```
 1:  package main
 2:
```

```
 3:  import (
 4:      "fmt"
 5:      "time"
 6:  )
 7:
 8:  func main() {
 9:      c := time.Tick(5 * time.Second)
10:      for t := range c {
11:          fmt.Printf("The time is now %v\n", t)
12:      }
13:  }
```

TRY IT YOURSELF ▼

使用 ticker 来显示当前时间

在这个示例中，您将明白如何使用 ticker。

1. 在终端中切换到文件夹 hour23/。
2. 在终端中执行命令 go run example04。
3. 您将看到终端中每隔 5s 就会重新显示当前时间。

▲

23.5 以字符串格式表示时间

不同计算机上的时间差别很多，使用字符串表示时间时，也有多种不同的方式。表 23.1 列出了不同的时间标准及其字符串表示。

表 23.1 时间的字符串表示

类型	字符串
ANSIC	Mon Jan _2 15:04:05 2006
UnixDate	Mon Jan _2 15:04:05 MST 2006
RubyDate	Mon Jan 02 15:04:05 -0700 2006
RFC822	02 Jan 06 15:04 MST
RFC822Z	02 Jan 06 15:04 -0700
RFC850	Monday, 02-Jan-06 15:04:05 MST
RFC1123	Mon, 02 Jan 2006 15:04:05 MST
RFC1123Z	Mon, 02 Jan 2006 15:04:05 -0700
RFC3339	2006-01-02T15:04:05Z07:00
RFC3339Nano	2006-01-02T15:04:05.999999999Z07:00

通常，使用上述标准将日期存储在数据库中，并以字符串的方式提供它们。Go 语言支持时间标准，也支持定义不符合这些标准的时间格式。

在程序清单 23.5 中，将一个使用 RFC3339 格式的时间字符串传递给了结构体 Time，再

将这个结构体打印到终端。

程序清单 23.5 分析表示时间的字符串

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "log"
 6:      "time"
 7:  )
 8:
 9:  func main() {
10:      s := "2006-01-02T15:04:05+07:00"
11:      t, err := time.Parse(time.RFC3339, s)
12:      if err != nil {
13:          log.Fatal(err)
14:      }
15:      fmt.Println(t)
16:  }
```

如果您运行这个示例，将发现成功地分析了这个表示时间的字符串。

```
go run example05.go
2006-01-02 15:04:05 +0700 +0700
```

TRY IT YOURSELF

分析表示时间的字符串

在这个示例中，您将明白如何分析表示时间的字符串。

1. 在终端中切换到文件夹 hour23/。
2. 在终端中执行命令 go run example05。
3. 您将发现字符串被分析为结构体 Time。

23.6 使用结构体 Time

将表示时间的字符串分析为结构体 Time 后，就可使用很多方法来使用它。程序清单 23.6 演示了结构体 Time 的一些方法。

程序清单 23.6 结构体 Time 包含的方法

```
 1:  package main
 2:
 3:  import (
 4:      "fmt"
 5:      "log"
 6:      "time"
 7:  )
 8:
 9:  func main() {
```

```
10:       s := "2006-01-02T15:04:05+07:00"
11:       t, err := time.Parse(time.RFC3339, s)
12:       if err != nil {
13:           log.Fatal(err)
14:       }
15:       fmt.Printf("The hour is %v\n", t.Hour())
16:       fmt.Printf("The minute is %v\n", t.Minute())
17:       fmt.Printf("The second is %v\n", t.Second())
18:       fmt.Printf("The day is %v\n", t.Day())
19:       fmt.Printf("The month is %v\n", t.Month())
20:       fmt.Printf("UNIX time is %v\n", t.Unix())
21:       fmt.Printf("The day of the week is %v\n", t.Weekday())
22:   }
```

如果您运行这个示例，将了解到结构体 Time 的一些方法的作用。

```
go run example06.go
The hour is 15
The minute is 4
The second is 5
The day is 2
The month is January
UNIX time is 1136189045
The day of the week is Monday
```

23.7 时间加减

方法 Add 在既有时间的基础上加上一定的时间，您可将其结果赋给变量。

```
s := "2006-01-02T15:04:05+07:00"
t, err := time.Parse(time.RFC3339, s)
if err != nil {
    log.Fatal(err)
}
nt := t.Add(2 * time.Second)
```

方法 Sub 从既有时间中减去指定的时间，您也可将其结果赋给变量。

```
s := "2006-01-02T15:04:05+07:00"
t, err := time.Parse(time.RFC3339, s)
if err != nil {
    log.Fatal(err)
}
nt := t.Sub(2 * time.Second)
```

23.8 比较两个不同的 Time 结构体

我们经常需要确定一个事件发生在另一个事件之前、之后还是它们是同时发生的。为让您能够这样做，time 包提供了方法 Before、After 和 Equal。这些方法都比较两个 Time 结构体，并返回一个布尔值。程序清单 23.7 演示了如何使用这些方法来比较两个日期。

程序清单 23.7 比较结构体 Time

```
1:  package main
2:
3:  import (
4:      "fmt"
5:      "log"
```

```
 6:      "time"
 7:  )
 8:
 9:  func main() {
10:      s1 := "2017-09-03T18:00:00+00:00"
11:      s2 := "2017-09-04T18:00:00+00:00"
12:      today, err := time.Parse(time.RFC3339, s1)
13:      if err != nil {
14:          log.Fatal(err)
15:      }
16:      tomorrow, err := time.Parse(time.RFC3339, s2)
17:      if err != nil {
18:      log.Fatal(err)
19:      }
20:      fmt.Println(today.After(tomorrow))
21:      fmt.Println(today.Before(tomorrow))
22:      fmt.Println(today.Equal(tomorrow))
23:  }
```

如果您运行这个示例，将发现它对时间的评估是正确的。

```
go run example07.go
false
true
false
```

▼ TRY IT YOURSELF

比较结构体 Time

在这个示例中，您将明白如何比较时间。

1. 在终端中切换到文件夹 hour23/。
2. 在终端中执行命令 go run example07。
3. 将对时间进行比较，并终端打印 true 或 false。

▲

23.9 小结

本章介绍了 Go 语言中的 time 包。您知道决定时间的变数有很多，包括时区、地区性调整，用户还可随便设置时间；您学习了网络时间协议——一种在计算机之间同步时间的方式；接下来，您学习了各种设置时间格式的标准，而 Go 语言能够分析其中的很多格式；您学习了如何让程序休眠、如何使用 ticker 以及结构体 Time 中用于读取数据、加减时间以及比较结构体 Time 的方法。

23.10 问与答

问：可以相信计算机上的时间吗？

答：正如您在本章看到的，计算机上的时间很容易被设置错误。因此，您应该明白，计算机上的时间很可能是错误的。在计算机上，用户设置的时区可能是错误的，设置的时间也可能与实际时间相距甚远。在您控制的服务器上，务必安装诸如网络时间协议等服务。

问：该使用哪种时间格式？

答：很多人都为标准化时间做出了努力，即便在计算机出现前，格林尼治标准时间就被全球用作共同的时间参考点。UTC 使用的是 ISO 8601 格式，这种格式在网上得到了很好的支持。RFC3339 是一个 ISO 8601 扩展。这两种标准都是不错的选择，如果有疑问，可使用得到广泛支持的 ISO 8601 标准。

问：该将 UTC 还是 GMT 作为网络时间？

答：当前，计算机和互联网都将使用时间标准 UTC。GMT 实际上是一个时区，而从理论上说，时区会受时间调整的影响，而时间标准不会。虽然主流社会将 GMT 视为一种时间标准，但计算和科学领域当前使用的是 UTC。

23.11 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

23.11.1 小测验

1. 壁挂钟和单调时钟之间有何不同？
2. 如何消除影响计算机上时间的变数，以确保时间的一致性？
3. 让 Goroutine 休眠有何影响？这会导致其他 Goroutine 停止执行吗？

23.11.2 答案

1. 壁挂钟显示的时间受时区和时间调整的影响，并不适合用来度量时间。单调时钟是稳定的，适合用来度量时间。
2. 通过使用网络时间协议（NTP），可与网上的服务器同步时间，还可用来检查时间误差。
3. 让 Goroutine 休眠，将导致它暂停执行，即什么都不做，但其他 Goroutine 可继续执行。

23.12 练习

1. 编写一个程序，计算圣诞节过后多少天就是新年了。
2. 编写一个程序，显示圣诞节 1600h 后是在新年前还是新年后。
3. 编写一个程序，将当前时间与您的出现时间进行比较，以显示您的年龄。

第 24 章

部署 Go 语言代码

本章介绍如下内容。

- 理解目标（**target**）。
- 压缩二进制文件的大小。
- 使用 **Docker**。
- 下载二进制文件。
- 使用 **go get**。
- 通过包管理器发布代码。

此时，您可能已经编写了一些能与他人分享的软件了。本章首先介绍如何为不同的环境编译 Go 代码，还有如何最大程度地缩小二进制文件的规模。然后，您将学习如何将二进制文件放在 Docker 容器中，以及提供可下载的二进制文件时需考虑的安全因素。

24.1 理解目标

Go 的优点之一是可在众多不同的操作系统和体系结构中运行。操作系统是 Windows 和 macOS 等系统，而体系结构指的是用于运行程序的计算机处理器的体系结构。体系结构决定了处理器的计算速度以及支持的内存量。当前市面上的计算机大都是 64 位的，但有些环境通常是 32 位的。使用 Go 语言编程时，编写的代码只需做少量的修改甚至无须修改就可在最常见的平台中运行。

如果您不知道所处的环境是什么样的，但已经安装了 Go，则可使用命令 go env 来获悉有关操作系统和体系结构的详细信息。

```
go env
GOARCH="amd64"
GOBIN=""
GOEXE=""
```

```
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/home/go/go"
GORACE=""
GOROOT="/usr/lib/go"
GOTOOLDIR="/usr/lib/go/pkg/tool/linux_amd64"
GCCGO="gccgo"
CC="gcc"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/tmp/
go-build674078985=/tmp/go-build -gno-record-gcc-switches"
CXX="g++"
CGO_ENABLED="1"
PKG_CONFIG="pkg-config"
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
```

在这个示例中，计算机是 64 位的，且安装的是 Linux 系统。默认情况下，编译器将针对当前操作系统和体系结构进行编译。因此，如果您当前使用的是 64 位的 Linux 系统，且要将代码部署到 64 位的 Linux 服务器，则什么都不用做。然而，如果要为不同的操作系统和体系结构编译程序，就需要指出这一点。

TRY IT YOURSELF ▼

获悉所处的环境

在这个示例中，您将明白当前所处的环境，包括操作系统和体系结构。

1. 打开终端。
2. 在终端中执行命令 go env。
3. 您将在控制台中看到有关当前环境的信息。您能从中找出有关操作系统和体系结构的信息吗？

▲

要指定目标平台，可以环境变量的方式给编译器指定操作系统和体系结构。要指定操作系统，可使用环境变量 GOOS；要指定体系结构，可使用环境变量 GO。要为 32 位的 Linux 系统编译程序，可像下面这样做。

```
GOOS=linux GOARCH=386 go build example01.go
```

要为 64 位的 Windows 系统编译程序，可执行下面的命令。

```
GOOS=windows GOARCH=amd64 go build example01.go
```

无须位于 32 位 Linux 系统或 64 位 Windows 系统中就可编译用于这些平台的二进制文件。例如，编译用于 Windows 系统的二进制文件时，将生成一个可在 Windows 系统上运行的.exe 文件。

TRY IT YOURSELF

为不同的目标进行编译

在这个示例中，您将明白如何为不同的目标编译 Go 代码。

1. 在终端中，切换到本书代码示例中的文件夹 hour24/。

2. 使用命令 GOOS=windows GOARCH=amd64 go build example01.go 编译用于 64 位 Windows 系统的二进制文件。

3. 使用命令 GOOS=linux GOARCH=amd64 go build example01.go 编译用于 64 位 Linux 系统的二进制文件。

Go 支持很多不同的操作系统和体系结构组合，如表 24.1 所示。

表 24.1 Go 支持的平台和体系结构

平台	体系结构
android	arm
darwin	386
darwin	amd64
darwin	arm
darwin	arm64
dragonfly	amd64
freebsd	386
freebsd	amd64
freebsd	arm
linux	386
linux	amd64
linux	arm
linux	arm64
linux	ppc64
linux	ppc64le
linux	mips
linux	mipsle
linux	mips64
linux	mips64le
netbsd	386
netbsd	amd64
netbsd	arm
openbsd	386
openbsd	amd64

续表

平台	体系结构
openbsd	arm
plan9	386
plan9	amd64
solaris	amd64
windows	386
windows	amd64

Go 编程语言的优点之一是，支持大量的操作系统和体系结构组合，因此程序员可为这些不同的目标编译 Go 代码。

24.2 压缩二进制文件的大小

发布 Go 语言代码很简单：通过编译生成一个二进制文件，其中包含运行程序所需的一切，因此您无须考虑依赖的问题。在诸如 Node.js 和 Ruby 等语言中，要将应用程序部署到生产环境，必须组合所有的依赖，并发布一系列文件。使用诸如 Go 等编译型语言时，这个过程要简单得多。

假设第 1 章的 Hello World 程序已准备就绪，就可以发布了。为此，可编译这个程序以生成一个二进制文件。

```
go build example01.go
```

从这个命令的输出可知，这里创建了一个二进制文件。

```
-rwxr-xr-x 1 go go 1.5M Sep 2 15:37 example01
-rw-r--r-- 1 go go   73 Sep 2 15:37 example01.go
```

对于这个将一行内容打印到控制台的简单程序，其生成的二进制文件为 1.5MB。怎么会这样呢？因为它包含执行这个程序所需的一切，其中包括 Go 语言 run-time。使用诸如 Ruby 和 Node.js 等语言时，由于 run-time 安装在服务器上，因此只需发布需要执行的文件。虽然这个二进制文件看起来较大，但优点是它不需要任何依赖，就能在编译时指定的系统中运行。

通过指定一些编译标志，可压缩编译得到的二进制文件的大小。这些标志指定省略符号表、调试信息和 DWARF 符号表。您无须理解这些省略的东西，只需知道发布程序时不需要它们就够了。

```
GOOS=linux go build -ldflags="-s -w" example01.go
```

使用这些标志进行编译时，二进制文件缩小到了 1001KB，这只有原来的三分之二！

```
-rwxr-xr-x 1 go go 1001K Sep 2 15:50 example01
-rw-r--r-- 1 go go    73 Sep 2 15:37 example01.go
```

在网络连接超过 100MB 的时代，发布 1001KB 的文件是可以接受的，但这个文件的

大小依然是个问题，可使用另一种技术进一步压缩它。在硬件设备很小，如为 Raspberry Pi、路由器或物联网（IoT）设备时，这种技术很有用。这些设备的存储空间通常有限，在这种情况下，可使用工具 upx。在 Linux 系统中，通常使用包管理器就能安装它。它对二进制文件进行压缩，这意味着运行时必须解压缩。虽然解压缩的速度很快，但这意味着启动时间会稍长些。通过对这个 1001KB 的二进制文件运行工具 upx，可进一步压缩小文件的大小。

```
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2017
UPX 3.94        Markus Oberhumer, Laszlo Molnar, & John Reiser   May 12th 2017
        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
    1024512 -> 382592     37.34%   linux/amd64   example01
Packed 1 file.
```

使用 upx 压缩后，Hello World 程序的大小被压缩到只有 374KB。对原本为 1.5MB 的程序来说，这样的效果很不错！

```
-rwxr-xr-x 1 go go 374K Sep 2 16:00 example01
-rw-r--r-- 1 go go   73 Sep 2 15:37 example01.go
```

压缩二进制文件的大小有多重要呢？这取决于要将代码部署到的环境。如果要通过网速很高的基础设施部署到 Web 服务器，20MB 的二进制文件也不是问题。如果要将固件部署到存储空间有限的设备，固件的大小就很重要了。如果您发布将通过包管理器下载的命令行工具，文件大小的重要性将适中。

总之，对大多数程序员来说，并不需要明白编译器的复杂性以及生成的文件很大的原因。在大多数情况下，花时间去压缩二进制文件的大小只会让人分神，直接发布默认生成的二进制文件就好了。Go 语言设计小组正在不断努力以压缩二进制文件的大小，每推出一个新版本，二进制文件的大小都得到了进一步压缩。

24.3 使用 Docker

Docker 是一种在虚拟机中运行应用程序的流行方式，它提供了一种轻量级的方式，可确保无论计算机使用的是哪种操作系统，应用程序都将在相同的环境中运行。Docker 使用容器来确保应用程序位于操作系统的沙箱环境中，开销比完整的虚拟机小。

最近几年，Docker 和容器越来越受欢迎。它们提供了确保一致性的轻量级方式，由于它们能够以标准方式运行，因此出现了一个以它们为中心的丰富而生机勃勃的生态系统。当前，大多数云服务提供商都提供了基于容器的服务，可托管和运行 Docker 容器。这种标准化催生了大量与容器相关的工具，让您能够轻松地对代码进行打包和发布。您可能听说过术语“开发和运维”，这指的是帮助软件小组快速、安全、平稳地发布代码。通过与其他工具结合起来使用，Docker 可帮助您完成这个过程。

通过使用 Docker，可建立部署管道，这意味着开发人员只需将代码提交到仓库，几分钟后它们就将奇迹般地自动发布到生产环境。

由于 Go 二进制文件包含运行程序所需的一切，因此有些人可能认为使用 Docker 来发布

Go 项目是小题大做。当然，就确保一致性而言，确实如此。Go 二进制文件要么在正确的操作系统和体系结构中执行，要么不是这样。但考虑到出现了大量帮助 Docker 发布容器的工具，应该使用 Docker 来打包并发布 Go 代码。

Docker 可用于 Linux、Windows 和 macOS。在 Linux 系统中可通过包管理器来安装，同时有针对大多数平台的可下载文件。

Go 官网维护着一些 Docker 映像，让您能够编译代码并在 Docker 容器中运行它们。这些映像主要用于在自动化环境中测试代码，但也可用于部署代码。这里不详细介绍 Docker 及其用法，而假定有一个简单的 Go Web 服务器已准备就绪，可部署到生产环境。这个简单的 Web 服务器在收到请求时响应以 Hello World，如程序清单 24.1 所示。

程序清单 24.1　示例 Web 服务器

```
 1:  package main
 2:
 3:  import "net/http"
 4:
 5:  func helloWorld(w http.ResponseWriter, r *http.Request) {
 6:      w.Write([]byte("Hello World.\n"))
 7:  }
 8:
 9:  func main() {
10:      http.HandleFunc("/", helloWorld)
11:      http.ListenAndServe(":8000", nil)
12:  }
```

程序清单 24.2 是一个 Docker 文件，它将这些 Go 代码复制到容器中，对其进行编译并暴露一个端口。您可在本地计算机上运行这个映像、将其推送到远程 Docker registry 或在支持 Docker 的基础设施上运行它。

程序清单 24.2　Docker 文件

```
1:  FROM golang:1.9
2:  COPY example02.go /
3:  RUN go build -o /example02 /example02.go
4:  EXPOSE 8000
5:  ENTRYPOINT ["/example02"]
```

要在本地计算机上构建这个映像，可执行如下命令。

```
docker build -t hello-go .
```

构建这个映像时，Docker daemon 将生成一些输出。

```
Sending build context to Docker daemon 3.072kB
Step 1/5 :  FROM golang:1.9
---> 5e2f23f821ca
Step 2/5 :  COPY example02.go /
---> Using cache
---> 121fb5f77ae8
Step 3/5 :  RUN go build -o /example02 /example02.go
---> Using cache
---> 319fffb93040
Step 4/5 :  EXPOSE 8000
---> Running in dc43db2be7f1
---> fd638d7376d5
```

```
Removing intermediate container dc43db2be7f1
Step 5/5 :  ENTRYPOINT /example02
---> Running in f0325da83631
---> 9807df129deb
Removing intermediate container f0325da83631
Successfully built 9807df129deb
Successfully tagged hello-go:latest
```

构建映像后，就可在本地计算机上运行容器了。

```
docker run -p 8000:8000 hello-go:latest
```

这个命令将容器的端口 8000 绑定到主机，这意味着可以访问这个容器了。如果您在浏览器中输入 http://localhost:8000/，这个应用程序将响应以文本 Hello World。

要停止这个应用程序，可再打开一个终端，并在其中执行如下命令。

```
docker ps
```

这将显示这个应用程序的 id 号。通过执行下面的命令，并指定这个 id 号，可关闭相应的 Docker 容器。

```
docker stop <container_id>
```

24.4 下载二进制文件

由于二进制文件提供了在平台上运行所需的一切，因此用户常常通过网络提供可下载的文件来分发 Go 项目。用户可根据其环境选择正确的文件，再下载并运行它。这是一种简单而轻量级的程序分享方式，因为将文件上传到互联网上进行分享易如反掌。

Watch Out!

警告： 通过互联网分发和下载文件。
这种方法根本无法保证代码名副其实、未被篡改。例如，攻击者很容易将一些代码放在网上，并允许任何人下载。而当您下载并运行这些代码时，所有硬盘数据都将被删除。

为证明可下载的文件名副其实，有两种常用的方法：一是将文件托管到有证书能够证明其身份的 https 网站，这让证书签发机构能够在用户连接到网站时验证其所有者；第二种常用的方法是提供文件的校验和，这个校验和相当于文件的指纹。

对于任何文件，都可使用相关的校验和查看器来查看其校验和。下面来编译代码，并查看生成的文件的校验和。假设您要通过互联网分发程序清单 24.1 所示的简单 Hello World 程序，生成二进制文件后，就可查看其校验和。

```
go build example03.go
sha1sum example03
40abf828e2c873dd7f57d91e4cc30cb923d5486f example03
```

如果您使用的是 macOS 系统，应使用 shasum 命令；如果您使用的是 Windows 系统，可下载文件校验和完整性验证器（File Checksum Integrity Verifiter）来验证校验和。

TRY IT YOURSELF

生成文件的校验和

在这个示例中，您将明白如何生成文件的校验和。

1. 在终端中，切换到本书代码示例中的文件夹 hour24/。
2. 在终端中，执行命令 go build example03.go。
3. 执行给二进制文件生成校验和的命令 sha1sum example03。
4. 您将在终端中看到打印出来的散列字符串。

校验和相当于独一无二的指纹，通过在下载网站随文件一起发布校验和，可在一定程度上保证文件就是您上传的文件。假设下载的服务器被攻击者攻破，并将文件替换为包含恶意代码的文件，如程序清单 24.3 所示。

程序清单 24.3 恶意的 Web 服务器

```
1:  package main
2:
3:  import "fmt"
4:  func main() {
5:      fmt.Println("Hello world!")
6:      fmt.Println("I am a hacker. I am going to delete everything.")
7:      fmt.Printf("%+v\n", u)
8:  }
```

那么编译这些代码，再为得到的文件生成校验和时，生成的散列值将完全不同。

```
go build example04.go
sha1sum example04
42ea77ae08ef8ef5d0dcf0a58ce61aaa0a618364 example04
```

对最终用户来说，这是严重的警告，千万不要运行这样的软件。如果您打算以可下载文件的方式分发 Go 软件，则务必提供校验和，让用户能够检查文件的完整性。另外，从互联网上下载 Go 二进制文件或其他软件时，务必检查提供的校验和，确认它与您为下载的文件而生成的校验和相同。

一个采用了这种做法的项目是 Terraform。这是一个使用 Go 语言编写的开源项目，它可以让用户配置云服务，以提供可重复的基础设施环境，用于部署代码。在 Terraform 下载页面，有用于各种平台和体系结构的二进制文件，还有列出这些可下载文件的校验和的链接。用户可下载二进制文件，并检查其校验和是否与发布的校验和相同。

24.5 使用 go get

在发布代码方面，一种轻量级但有效的方式是使用命令 go get。第 21 章介绍了如何安装第三方包，而命令 go get 也可用来安装命令行工具，官方采用的也是这种方法。这种低干涉

方法非常有效：您可将工具的源代码分享到 Github 等网站，而用户只需执行一个命令就能安装它。例如，要安装依赖管理工具 dep，可像下面这样做。

```
go get -u github.com/golang/dep/cmd/dep
```

安装这个包后，就可执行命令 dep。

```
dep-help
dep is a tool for managing dependencies for Go projects
```

go get 尤其适合用来分享命令行工具。它与 Go 工作流程紧密集成，并在需要审阅源代码或文档时带来另一个好处：本地有代码的副本。在第 19 章，您学习了如何以这种方式发布代码，这是另一种发布代码的方式。

24.6 通过包管理器发布代码

虽然通过互联网分享文件是一种简单的 Go 程序分享方式，但操作系统大多都有用于分发代码的官方或非官方包管理器，这让最终用户能够以一致的方式安装软件。很多包管理器还提供了其他安全功能，如自动比较校验和或要求开发人员对包进行数字签名。下面是一些流行的包管理器及其应用的平台。

- Homebrew：用于 macOS 系统。
- Chocolatey：用于 Windows 系统。
- Apt：用于 Debian 和 Ubuntu 系统。
- Yum：用于 Fedora、Red Hat 和 Centos 系统。
- Pacman：用于 Arch Linux 系统。

这些包都让最终用户能够以安全而一致的方式安装软件。如果您要将开源软件发布到更大的社区，推荐将其发布到包管理器中。

24.7 小结

本章介绍了一些分发 Go 程序的方式。您了解到，可为众多不同的平台和体系结构编译 Go 代码，这是 Go 编程语言的又一个优点；您学习了如何缩小二进制文件的规模，以及如何使用 upx 等工具来进一步压缩二进制文件；您了解到，Docker 是一种分享 Go 程序的方式，以及如何将二进制文件上传到互联网以分享它们；您学习了一些安全问题及其缓解方法；最后，您学习了包管理器，它们提供了安全而一致的软件安装方式。

24.8 问与答

问：该如何分享自己编写的软件？通过电子邮件发送、上传到网上供人下载还是使用 Docker？

答：这个问题没有唯一的答案。如果您要部署 Web 应用，使用 Docker 是不错的选择；如果您要分享命令行工具，将其上传到网上供人下载或使用 go get 是不错的选择；如果您编

写的是固件，则可能需要采用完全不同的分享方式。最重要的是文件的接收者能够验证其完整性。为此，发送或发布校验和是不错的选择，根据要在多大程度上保证文件名副其实，可能还需考虑对文件进行数字签名。

问：Go 二进制文件真的不需要依赖吗？

答：不需要。这是使用静态链接的二进制文件的优点之一。只要二进制文件是针对正确的目标编译而成的，运行它时就不需要其他任何东西。

问：Docker 看起来很复杂，真的值得去学习吗？

答：如果您只使用 Go 语言来创建命令行应用程序和系统工具，则可能不需要学习 Docker；如果您要创建服务器或将部署到其他应用环境中的软件，就绝对应该学习 Docker；如果您是软件小组的一员，并要部署 Web 产品，就很可能要用到 Docker。包括 Google 在内的很多大型软件公司都使用容器来管理基础设施。

24.9 作业

作业包含小测验和答案，旨在加深对本章内容的理解。请尽可能先回答所有的问题再看后面的答案。

24.9.1 小测验

1. 对于要在其中运行 Go 代码的硬件，如何获悉其体系结果？
2. 在 Linux 系统中，可编译生成能够在 Windows 系统上运行的二进制文件吗？
3. 该信任从网上下载的 Go 二进制文件吗？

24.9.2 答案

1. 如果机器上安装了 Go，就可使用命令 go env 来获悉大量有关该环境的信息，其中包括环境变量 GOOS 和 GOARCH。如果没有安装 Go，可使用操作系统提供的功能来获悉底层体系结构。至于使用的是哪种操作系统，只要开机就能知道。

2. 可以。这被称为跨平台编译。然而，在与目标平台相同的平台上编译时，速度通常更快！

3. 一般而言，不应信任从互联网上下载的二进制文件，但如果您能确定网站的身份，则可放心大胆地执行下载的文件。

24.10 练习

找一位这样的朋友：他使用的操作系统与您使用的不同。通过编译生成一个可在这位朋友的计算机上运行的二进制文件。在不同的操作系统之间传输文件时，使用校验和对文件进行验证。

Go语言入门经典

SAMS

Teach Yourself

通过阅读本书，你将学会：

- 通过Go开发工具和Web服务器快速提高效率；
- 掌握核心功能，包括字符串、函数、结构和方法；
- 使用类型、变量、函数和控制架构；
- 充分利用Go的数组、切片和地图功能；
- 通过Goroutine和通道编写功能强大的并发软件；
- 顺利处理程序错误；
- 通过包促进代码重用；
- 掌握Go高效编码的独特惯用法；
- 使用正则表达式和时间/日期函数；
- Go的测试和基准测试；
- 编写基本的命令行程序、HTTP服务器和HTTP客户端；
- 高效地将Go代码转化为产品；
- 构建基本的TCP聊天服务器和JSON API。

24章阶梯教学

通过阅读本书，读者将会掌握如何使用Go语言开发更灵活、更可靠和更具可扩展性的软件。本书采用直观、循序渐进的方法，引导读者掌握从设置开发环境到测试、部署解决方案在内的所有知识。本书还借助于实例介绍了Go语言的基本架构，演示了Go语言在并发和网络编程方面的突破性特性，并阐述了Go语言强大的编程惯用法。本书每章内容都建立在已学的知识之上，从而为读者理解并实现Go语言程序打下坚实的基础。

循序渐进的示例引导您完成最常见的Go任务。

问与答、测验和练习帮助读者检测知识的掌握情况。

“注意”、“提示”和“警告”指出捷径和解决方案。

George Ornbo是一位企业家、作家、博主和程序员，他帮助创办了包括pebble {code}、Seatwave、Bede Gaming和VotesForSchools等在内的初创公司，并为《卫报》撰写过技术方面的文章，还经常在技术博客上发表博文。

异步社区 www.epubit.com

异步社区 www.epubit.com
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

ISBN 978-7-115-48503-8

分类建议：计算机／程序设计／Go语言
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-48503-8
定价：59.00元